Automatic Testing of Software with Structurally Complex Inputs

by

Darko Marinov

B.S., University of Belgrade, Yugoslavia (1997)
S.M., Massachusetts Institute of Technology (2000)

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

February 2005

Author ....
Department of Electrical Engineering and Computer Science
December 14, 2004

Certified by ....
Martin C. Rinard
Associate Professor, Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Automatic Testing of Software with Structurally Complex Inputs

by

Darko Marinov

Submitted to the Department of Electrical Engineering and Computer Science
on December 14, 2004, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Modern software pervasively uses structurally complex data such as linked data structures. The standard approach to generating test suites for such software, manual generation of the inputs in the suite, is tedious and error-prone. This dissertation proposes a new approach for specifying properties of structurally complex test inputs; presents a technique that automates generation of such inputs; describes the Korat tool that implements this technique for Java; and evaluates the effectiveness of Korat in testing a set of data-structure implementations.

Our approach allows the developer to describe the properties of valid test inputs using a familiar implementation language such as Java. Specifically, the user provides an imperative predicate—a piece of code that returns a truth value—that returns true if the input satisfies the required property and false otherwise. Korat implements our technique for solving imperative predicates: given a predicate and a bound on the size of the predicate's inputs, Korat automatically generates the bounded-exhaustive test suite that consists of all inputs, within the given bound, that satisfy the property identified by the predicate. To generate these inputs, Korat systematically searches the bounded input space by executing the predicate on the candidate inputs. Korat does this efficiently by pruning the search based on the predicate's executions and by generating only nonisomorphic inputs.

Bounded-exhaustive testing is a methodology for testing the code on all inputs within the given small bound. Our experiments on a set of ten linked and array-based data structures show that Korat can efficiently generate bounded-exhaustive test suites from imperative predicates even for very large input spaces. Further, these test suites can achieve high statement, branch, and mutation coverage. The use of our technique for generating structurally complex test inputs also enabled testers in industry to detect faults in real, production-quality applications.

Thesis Supervisor: Martin C. Rinard
Title: Associate Professor, Electrical Engineering and Computer Science

Acknowledgements

Numerous people have helped me, directly or indirectly, to complete this work. With the previous sentence, I started the acknowledgements for my bachelor's and master's theses. The list of people has grown even longer for this PhD thesis. I apologize in advance to those who I will not explicitly mention here due to the lack of space or the negative influence of PhD studies on my memory. You all have my gratitude for your help.

First of all, I would like to thank my advisor, Martin Rinard, for providing me the necessary support—personal, financial, and technical—throughout my graduate studies. Martin's care for the overall well-being of his students is truly great. I was extremely fortunate to be able to rely on him in the many uncommon situations that I faced as an MIT student. (Getting married and having a baby, for example, sound common compared to having your home country bombed or being questioned by the police and calling your advisor at home at 3 a.m.) I also appreciate the tremendous freedom that he gave me in initiating and producing my research and publishing the results. Although it meant that we did not work very closely, and I had to learn some lessons indirectly, it still helped me to learn those lessons well. Finally, I thank Martin for believing in me ever since he brought me to MIT. I hope that I fulfilled his expectations and that I can continue to do so as he raises the expectation bar.

I would like to thank Michael Ernst and Daniel Jackson for all their help during my graduate studies. They served on my thesis committee and helped me improve the writing of this dissertation. They also helped me with their advice and discussions on selecting research topics and doing research. I wish I had time to collaborate more with them.

Sarfraz Khurshid helped the most with the work presented in this dissertation. Sarfraz and I started working together on automated testing of software with structurally complex inputs while we were roommates in the infamous apartment 001 in the Edgerton dorm. Our initial, half-serious development of a testing tool called TestEra grew into the whole MulSaw project that Sarfraz and I co-led for about three years, working with a dozen people, producing research that resulted in eight publications, and having a lot of fun. From the MulSaw project, Sarfraz and I also split our PhD dissertations, his on the ideas behind TestEra and mine on the ideas behind Korat. Sarfraz was not only an excellent collaborator but also an excellent friend throughout my MIT studies, from the time we met at the prospective student visit weekend when I first came to MIT through my PhD defense that he came from Texas to attend. Without Sarfraz, my experience at MIT wouldn't have been nearly as positive.

The PhD students in Martin's group—C. Scott Ananian, Chandrasekhar Boyapati, Brian Demsky, Viktor Kuncak, Patrick Lam, Maria-Cristina Marinescu, Radu Rugina, Alexandru Salcianu, and Karen Zee—made my life at MIT a pleasant experience and helped me in various ways during my graduate studies. Chandra helped most directly with my thesis: his question during one of my presentations led to the idea of Korat; the name *Korat* actually comes from Chandra's native language and means *saw* (the tool for cutting wood). Maria made my first office the most productive place that I have worked at so far. Radu had enough patience to teach me several technical

topics at the beginning of my PhD studies. Viktor and Alex had the patience to discuss research with me, either specific technical questions or meta-questions about doing research, throughout our PhD studies. Brian and Patrick provided additional technical insights, and Scott helped me make my working environment pleasant. Last but not least, Karen provided guidance on proper behavior.

Several other MIT students helped me with technical discussions or general advice about research and MIT. Alan Donovan, Carlos Pacheco, Derek Rayside, Robert Seater, Ilya Shlyakhter, and Manu Sridharan provided useful feedback on my work. Kostas Arkoudas helped me with discussions on theorem proving and mathematical rigor. Eddie Kohler and Jonathan Babb were the two senior PhD students who most influenced my decisions to come to MIT and to stay at MIT. Sam Larsen, Dim Mitsouras, and Mark Stephenson started the PhD program with me, and I wish I could have spent more time with them.

I was fortunate to work, during my graduate studies, with several undergraduate and MEng students, including Omar Abdala, Daniel G. Adkins, Basel Al-Naffouri, Alexander Andoni, Faisal Anwar, Suhabe Bugrara, Cristian Cadar, Dumitru Daniliuc, Ang-Chih Kao, Ioan Tudor Leu, Jelani Nelson, and Lev Teytelman. This experience taught me how (not) to supervise students. I would especially like to thank Alex and Dumi for their help with the experiments on Korat. Alex developed most of our mutation testing framework, and Dumi developed various scripts.

My two summer internships were very valuable and pleasant experiences because of the people I worked with. At IBM Research, I worked with Rob O'Callahan in a group managed by John Field. Rob is an amazing hacker, and I was fortunate to be able to learn several hacks from him. At Microsoft Research, I worked with Wolfram Schulte in a group managed by Yuri Gurevich. Wolfram turned the research ideas behind Korat into a part of a practical tool used by real testers and enabled me to learn how testing is done "in the trenches." I also thank Wolfgang Grieskamp and Nikolai Tillmann for discussions on Korat.

Toward the end of my graduate studies, I collaborated with Tao Xie and David Notkin from the University of Washington. Tao made our collaboration as productive and enjoyable as the best collaborations I had at MIT, although we worked together mostly long-distance. David accepted me as a close collaborator and provided valuable advice on several matters. I thank them for enriching my PhD experience beyond the usual level.

Jovan Djordjevic, Davor Magdic, Aleksandar Milenkovic, Dragan Milicev, Veljko Milutinovic, Jelica Protic, and Igor Tartalja provided me with important extracurricular activities and initial research opportunities during my undergraduate studies. Without them, I would have not been at MIT in the first place.

I thank Mihai Budiu for all the jokes and serious emails that he sent me in the last few years. I am truly sorry that a joke about August never became reality.

Mary McDavitt helped me tremendously with organizing my thesis defense and handling other administrative issues of my PhD. There are not enough words or flowers to thank her.

I would like to thank many friends who helped me remain (in)sane during my stay at MIT. Their support and friendship enabled me to put my work in perspective with other life issues. The friends from the U.S.A. include Vladimir Bozin, Cindy Chen, Ivan Crnilovic, Jure Derganc, Milenko Drinic, Ilse Evans, Jim Evans, Vladimir Jakobac, Nikola Jevtic, Ivana Kalea, Aleksandar Kojic, Bozidar Kuljic, Gojko Lalic, Marko Loncar, Tatjana Marvin, Jelena Mirkovic, Nebojsa Mirkovic, Mrdjan Mladjan, Milos Popovic, Milica Radisic, Zoran Radisic, Rados Radoicic, Sasa Slijepcevic, Adnan Sulejmanpasic, and Bosiljka Tasic. Of the friends who are not (yet) in the U.S.A., I would like to mention Vladimir Baltic, Goran Maksimovic, Zoran Martic, and Nenad Stamenkovic, who made it to my wedding, as well as Bosko Nikolic, who could not make it.

Danica Galonic, Ting Hsiao, Vladimir Lukic, Nemanja Petrovic, and Scott Yih helped my family while we were moving from Massachusetts to Illinois.

My research assistantships were funded in part by ITR grant #0086154 from the National Science Foundation.

Last but not least, I would like to thank my sister, Marina, my mother, Ivanka, and my father, Slobodan, for their enormous love, care, and support. I will strive to give the same to my daughter, Lara, and my wife, Daniela; they deserve much more for the patience and understanding they show for me and my work.

Contents

1 Introduction 13

1.1 Background 15

1.1.1 Imperative Predicates as Constraints 16

1.1.2 Korat: A Solver for Imperative Predicates 17

1.1.3 Eliminating Equivalent Test Inputs 18

1.2 Bounded-Exhaustive Testing 19

1.2.1 Evaluation 20

1.3 Thesis and Challenges 21

1.4 Contributions 23

1.5 Organization 24

2 Examples 25

2.1 Binary Search Tree 25

2.1.1 Preview of the Experimental Results 27

2.2 Heap Array 29

3 Solving Imperative Predicates 33

3.1 Problem Statement 33

3.1.1 Definitions 33

3.1.2 Formal Correctness Statement 36

3.1.3 Naive Algorithm 37

3.1.4 Predicates with Multiple Inputs 38

3.2 Korat: A Solver for Imperative Predicates 39

3.2.1 Overview 39

3.2.2 Finitization 40

3.2.3 State Space 42

3.2.4 Search 45

3.2.5 Nonisomorphism 47

3.2.6 Implementation 49

4 Correctness 53

4.1 Challenges 53

4.2 Notation 54

4.3 Conditions 54

4.4 Statement 56

4.5 Proof 56

5 Direct Generation **61**

5.1 Example 61

5.2 Comparison with Imperative Predicates 63

5.3 Dedicated Generators 65

6 Testing Based on Imperative Predicates **67**

6.1 Unit Testing 67

6.1.1 Overview of Java Modeling Language 67

6.1.2 Black-Box Method Testing 69

6.1.3 White-Box Method Testing 72

6.1.4 Sequences of Method Calls 72

6.2 System Testing 74

6.2.1 Differential Testing 74

7 Mutation Testing **77**

7.1 Overview 77

7.1.1 Output Comparison 78

7.1.2 Existing Tools 78

7.2 Ferastrau: A Mutation Tool for Java 79

7.2.1 Mutant Generation 79

7.2.2 Mutant Execution 80

8 Evaluation **83**

8.1 Benchmarks 83

8.2 Generation of Data Structures 85

8.2.1 Dedicated Generators 87

8.3 Bounded-Exhaustive Testing 88

8.3.1 Mutation 89

8.3.2 Coverage 89

8.3.3 Feasibility 92

8.3.4 Randomness 94

9 Comparison with Testing Based on Declarative Predicates **97**

9.1 Overview of TestEra 97

9.2 Ease of Writing Predicates 98

9.2.1 Nonisomorphic Inputs 99

9.3 Efficiency of Generation and Checking 101

9.4 Summary 103

10 Related Work **105**

10.1 Automated Test-Input Generation 105

10.2 Static Analysis 107

10.3 Software Model Checking 108

11 Conclusions **111**
11.1 Discussion 111
11.1.1 Language for Predicates 111
11.1.2 Efficiency of Generation 112
11.1.3 Bounded-Exhaustive Testing 114
11.2 Future Work 115
11.3 Summary 117

A Full Example Code and Experimental Results **119**

List of Figures

2-1 Binary search tree example. 26
2-2 Trees that Korat generates for scope three 28
2-3 Variation of code coverage and mutating coverage with scope 29
2-4 Heap array example. 30
2-5 Heaps that Korat generates for scope one. 31

3-1 Example object graphs that are not binary search trees 34
3-2 Example (non)isomorphic object graphs 35
3-3 Algorithm for checking isomorphism of two object graphs. 37
3-4 Example of a predicate with multiple inputs 38
3-5 High-level overview of the Korat search algorithm 39
3-6 Two finitizations for the `repOk` method. 40
3-7 Classes that Korat uses to represent state space. 41
3-8 State space for `finSearchTree(3)`. 42
3-9 Candidate that is a valid `SearchTree`. 43
3-10 Candidate that is not a valid `SearchTree`. 43
3-11 Pseudo-code of the Korat's search algorithm. 46
3-12 Instrumentation for the `SearchTree` example. 50

5-1 A noncontrollable generator for trees. 62
5-2 A controllable generator for binary search trees. 63

6-1 Example JML specification for the `remove` method. 68
6-2 Example JML specification for the `extractMax` method. 69
6-3 Predicate for black-box testing of the `remove` method. 70
6-4 Finitizations for the precondition of the `remove` method. 71
6-5 Comparison of several testing frameworks for Java 72
6-6 Predicate for white-box testing of the `remove` method. 73
6-7 Methods corresponding to the example method sequences. 74
6-8 Differential testing of an XPath compiler. 75

8-1 Benchmarks and finitization parameters 84
8-2 Performance of Korat for generation of data structures 86
8-3 Target methods for testing data structures 88
8-4 Testing target methods with bounded-exhaustive test suites 90
8-5 Variation of code coverage and mutating killing with scope 91

8-6 Performance of Korat for generating test inputs 93
8-7 Comparison of bounded-exhaustive testing and random testing 94

9-1 Example declarative predicate in Alloy 98
9-2 Manual symmetry-breaking predicate in Alloy 100
9-3 Comparison of Korat and SAT 102

A-1 Full code for the example `remove` method. 119
A-2 Full code for the example `repOk` method. 120
A-3 Korat's performance for test generation and checking 121

Chapter 1

Introduction

Testing is currently the dominant method for finding software errors. As such, testing is critical to the production of high-quality code. Obtaining good test inputs is a key requirement for successfully testing any software system. Manual generation of test inputs is a labor-intensive task and typically produces inputs that exercise only a subset of the functionality of the software. The alternative is to automate generation of test inputs; such automation can significantly help the developers produce and maintain reliable code.

We focus on software components that have *structurally complex* inputs: the inputs are structural (e.g., represented with linked data structures) and must satisfy complex properties that relate parts of the structure (e.g., invariants for linked data structures). Our main contribution is a technique for the automatic generation of structurally complex inputs from input properties provided by the developer. We assume that the developer also provides a test oracle that can automatically check partial correctness of the software for every input.

Modern software pervasively manipulates structurally complex data. For example, Java programs manipulate a heap that consists of linked objects; each heap configuration must satisfy the consistency properties of the data structures in the heap. As another example, Web services manipulate XML documents [114]; each service operates only on documents that satisfy particular syntactic and semantic properties. We next present two scenarios that illustrate the testing of code that takes structurally complex inputs, and we discuss the need for generation of such inputs.

Consider the testing of an operation of a data structure that implements an abstract data type, for example, a Java method that removes an element from a tree that implements a set. Each such operation should preserve the *invariant* of the data structure; in our example, the invariant is that the object graph that represents the tree is indeed a tree and not an arbitrary graph with cycles. To test the remove method, we need to execute it on several input object graphs. This method has an implicit precondition: the input must be a tree. We thus do not want to test the remove method for arbitrary object graphs—it may very well loop infinitely or return an incorrect output; we need to generate object graphs that satisfy the tree invariant.

As another example, consider the testing of a program that processes XML documents of some specific kind. The input to the program is simply a sequence of characters; if the sequence does not satisfy a syntactic or semantic property of the XML documents that the program processes, the program only reports an error. It is important to generate test inputs that check this error-reporting behavior. But we also need to check the actual processing, which requires generating test inputs that satisfy the properties of the kind of XML documents that the program processes.

One potential approach to the automatic generation of structurally complex inputs is random generation: automatically generate a set of inputs, then filter out those inputs that do not satisfy the required property of the component under test. However, this approach may be prohibitively expensive for components with complex input properties—the density of inputs that satisfy the property in the search space may be so low that the generator is unable to produce enough legal inputs within a reasonable amount of time.

We propose a systematic approach for generating structurally complex inputs by *solving imperative predicates.* An imperative predicate is a piece of code that takes an input, which we call a *structure*, and evaluates to a boolean value. Solving an imperative predicate amounts to finding structures for which the predicate evaluates to true. An efficient solver for imperative predicates can form the basis of a practical tool for test-input generation. Specifically, the user represents test inputs with structures and provides an imperative predicate that checks the desired property: the predicate returns true if and only if the structure satisfies the property. The tool then processes this predicate to efficiently produce a stream of structures that satisfy the property identified by the predicate; each structure translates into a test input.

We have developed a technique for solving imperative predicates and implemented it in a solver called Korat.[1] In our technique, a solver takes as input an imperative predicate and additionally a *finitization* that bounds the size of the structures that are inputs to the predicate. The solver systematically searches the bounded structure space and produces as output *all* structures, within the given bounds, for which the predicate returns true. More precisely, the solver produces only nonisomorphic structures that translate into a special kind of nonequivalent test inputs (Section 1.1.3).

We have used *bounded-exhaustive testing* [56,72,104], which tests the code for all inputs within the given small bound, to evaluate the effectiveness of our technique for generating structurally complex test inputs by solving imperative predicates. More precisely, we have used Korat to generate bounded-exhaustive test suites for ten data-structure implementations, half of which we developed following standard textbook algorithms [23] and half of which we took from other sources, including the Java standard library [105]. The results show that Korat can efficiently generate test suites that achieve high statement, branch, and mutation coverage (Section 1.2.1).

The rest of this chapter is organized as follows. Section 1.1 presents the use of constraint solving for generation of test inputs and discusses imperative predicates as a particular kind of constraint. Section 1.2 presents our approach to testing based on solving imperative predicates and describes how we evaluated this approach.

[1] We initially [13] used the term *Korat* to refer to the solving technique itself, to a solver that implements this technique for Java predicates, and to an application of this solver to black-box, specification-based unit testing. In this dissertation, we use *Korat* to refer only to the solver.

Section 1.3 states the main thesis of this dissertation, and Section 1.4 lists the contributions that the dissertation makes. Section 1.5 presents the organization of the remainder of this dissertation.

1.1 Background

We next discuss the use of constraint solving for test-input generation. A *constraint* is a formula that specifies some property of a set of variables. For example, the constraint `x >= 1 && x <= 8` specifies that the value of the variable x be between one and eight. Another familiar example constraint is from the "eight queens problem"; it specifies that a chess board contain eight queens such that no queen attacks another. *Constraint solving* amounts to finding solutions to constraints, i.e., assignments to the variables that make the constraint formula true. For example, `x = 3` is one solution for the first example constraint. Constraint solvers often use sophisticated local and global search techniques to find one, several, or all solutions to a given constraint.

Researchers have explored the use of constraint solving for test-input generation since the 1970s [29,39,49,50,61,63,68,73]. The main idea is to represent a property of the desired test inputs as a constraint (potentially obtained automatically from the code under test) such that a solution to the constraint translates into a test input. For example, a function that draws a nonstandard chess board may take as input the board size—an integer that is between one and eight. The first example constraint represents this condition on the input; any solution to this constraint gives a value to x that constitutes a legal input for this function.

Our approach allows the tester to provide the constraints for test inputs. Note that these constraints characterize the "desired" inputs, which need not always be the "legal" inputs to the code. For example, the tester may want to test the chess-board-drawing function with a board size that is not between one and eight. Instead of providing specific values for the board size, the tester provides only the constraint, and a constraint solver then generates inputs. This is indeed the main advantage of using constraint solvers: the tester needs to provide only the constraint, which may be much easier to write than a set of test inputs, and a tool then automatically generates a set of test inputs.

Three questions arise in using constraint solving for test-input generation:

1. Which language should we use to express constraints, and what type of constraints can we express in that language?

2. What constraint solver should we use to generate test inputs?

3. How can we eliminate equivalent test inputs?

Most previous approaches focused on constraints where variables are integers or arrays of integers [29, 50, 61, 63]. These approaches did not consider generation of structurally complex inputs. Although it is in principle possible to encode structural inputs (e.g., an object graph on the Java heap) using integer arrays, expressing relevant properties of these inputs (e.g., that a part of the object graph is acyclic) would be very cumbersome. Furthermore, the previous studies did not optimize solvers for efficient generation of all solutions for such constraints.

We first developed an approach [55, 73] for generating structurally complex test inputs from constraints written in a declarative, relational language based on first-order logic formulas [52]. (More details on this approach are in Chapter 9.) This constraint language enables the succinct expression of relevant structural properties such as acyclicity. We used the existing solver for this language [51] and developed a methodology for eliminating equivalent test inputs [57]. Based on this approach, we implemented a testing tool; the use of this tool detected faults in several real applications [56, 73, 104]. However, this approach requires the user to learn a constraint language that is quite different from a typical programming language. We therefore developed a new approach, which is the topic of this dissertation. Our new approach allows the user to write properties of test inputs as imperative predicates in a familiar programming language such as Java.

1.1.1 Imperative Predicates as Constraints

An imperative predicate is a piece of code that takes an input *structure* and determines its validity. In Java, for example, an imperative predicate is a method that returns a boolean. We use the term *imperative predicates* for two reasons: (1) to determine whether the result is true or false for the given structure, a tool has to execute an imperative predicate that typically manipulates state; and (2) it distinguishes imperative predicates from the declarative predicates that we used in our previous approach. Solving an imperative predicate amounts to finding structures for which the predicate returns true; we call such structures *valid structures.*

Imperative predicates give the user an operational way of expressing declarative constraints that identify the desired test inputs. The user provides an imperative predicate, written in a standard programming language, that returns true if the input structure satisfies the required property and false otherwise; every valid structure translates into a test input.

Our approach involves an unusual design decision: the user expresses *declarative* constraints with *imperative* predicates. We next briefly compare our approach with two other approaches to generating all valid structures that represent structurally complex test inputs with the desired property: (1) an approach where the user writes constraints as declarative predicates, as in our previous approach, and (2) an approach where the user writes programs, called *generators*, that directly generate valid structures instead of checking their validity. Our comparisons consider the three main features of an approach to generating structures:

1. How easy is it for the user to develop machine-readable descriptions (constraints or generators) from which a tool can generate desired structures?

2. How efficient is the generation of such structures?

3. For which testing tasks are the descriptions useful?

Chapter 9 presents a detailed comparison of our approach, which is based on imperative predicates, with an approach based on declarative predicates. We do not have experimental results that compare the ease of development of these predicates, but our anecdotal experience shows that using the programming language in which the code was written to write predicates for test inputs has several advantages:

- Users are familiar with the syntax and semantics of the language, so there is no new language to learn.
- Users are familiar with the development tools, such as integrated development environments and debuggers, and can use them effectively to develop imperative predicates.
- Some imperative predicates may already be present in the code, typically in assertions for automatic checking of correctness.

Furthermore, the experimental results for our benchmarks show that generating structures from imperative predicates is faster than generating them from declarative predicates. However, declarative predicates tend to be more succinct than imperative predicates, and users familiar with a declarative language may find it easier to develop declarative predicates. Additionally, for test inputs that are not structurally complex, declarative predicates are typically easier to write and faster to solve than imperative predicates. Both imperative and declarative predicates are useful (with the support of appropriate tools) for the same testing tasks, including automatic test-input generation and use as automatic test oracles.

Chapter 5 presents a comparison of our approach with generators that directly generate valid structures. Our anecdotal experience shows that for simple properties it may be as easy to write generators (especially in languages that have built-in backtracking) as predicates, but for more complex properties it can be often easier to write predicates. Using optimized generators to generate structures is always faster than solving imperative predicates. However, optimized generators are harder to change for similar properties than predicates are. Moreover, predicates can serve as test oracles, whereas generators cannot.

In summary, our design decision seems to be the "sweet spot" for bounded-exhaustive generation of structures that represent structurally complex test inputs. But for a different type of generation or a different set of properties for test inputs, it is likely that other approaches are more appropriate.

1.1.2 Korat: A Solver for Imperative Predicates

The main contribution of this dissertation is a technique for solving imperative predicates and its implementation in Korat, a solver for Java predicates. Korat takes as input a Java predicate and a finitization that bounds the size of the input structures. Korat produces as output all valid nonisomorphic structures within the given bounds. We discuss isomorphism in the next section.

In Java, each structure is an object graph that consists of linked objects whose fields point to other objects or have values of primitive type. Each *finitization* specifies bounds for the number of objects in the structure and possible values for the fields of these objects. Each finitization thus defines a finite structure space—a set of all those object graphs that consist of up to the specified number of objects and whose fields have one of the specified values. We call the numeric bound that limits the size of the structures the *scope*. Whereas finitization describes the whole structure space, usually parameterized over size, the scope provides the actual numeric value for the bound on the size parameter.

Korat generates all valid structures by systematically searching the structure space defined by the finitization. A naive systematic search would enumerate every structure within the finitization and then check whether the predicate returns true or false for that structure. Korat uses an optimization that prunes the search. Specifically, Korat builds candidate structures and executes the predicate to check whether these structures are valid;[2] Korat *monitors* the execution of the predicate to determine the fields of the structure that the predicate accesses before returning the value. The values of the fields that the predicate does not access do not affect the result, so Korat eliminates from consideration all additional structures that differ only in the values of those fields. Note that this heuristic optimizes the efficiency of the search but does not compromise its completeness. Chapter 3 presents Korat in detail.

It is necessary to empirically evaluate how well Korat's pruning works. The efficiency of the pruning depends not only on the set of valid structures but also on the way the imperative predicate is written. For two predicates with the same set of valid structures, Korat can take a greatly different amount of time to generate the structures. More precisely, the efficiency depends on the order in which the predicate accesses the fields. In theory, an ill-written predicate might always read all the fields of the structure before returning a value, thereby forcing Korat to explore almost the entire structure space, like the naive systematic search. (Korat can still avoid exploration of isomorphic structures.) In practice, however, our experience indicates that naturally written predicates, which return false as soon as they detect a violation, induce very effective pruning. Section 11.1.2 presents several guidelines for writing predicates that make Korat's pruning effective and also discusses properties for which Korat is inefficient. The results in Chapter 8 show that Korat's performance is a significant improvement over that of the naive search; Korat can explore very large structure spaces in just a few seconds.

1.1.3 Eliminating Equivalent Test Inputs

We next discuss equivalent and isomorphic test inputs. Assume that we are testing a program by running it on several inputs and checking one correctness property for each output. Two test inputs are equivalent if they have the same behavior with respect to the correctness property—they either both make it succeed or both make it fail. In other words, either both test inputs detect a fault or neither of them detects a fault. The equivalence classes of such inputs form revealing subdomains [48,

[2] This search technique is known as *generate and test* [92], but we avoid this term because we use the terms *generate* and *test* for other purposes.

112]. In theory, if we could precisely partition the program's inputs into revealing subdomains, it would suffice to test the program with one representative from each revealing subdomain. In practice, however, it is hard to determine the revealing subdomains for a given program and a given correctness property.

Our generation focuses on *isomorphic inputs*, a special kind of equivalent inputs. Two test inputs are isomorphic for a set of programs and correctness properties if they have the same behavior for *all* programs and *all* properties from the set. The equivalence classes of isomorphic inputs thus form the finest revealing subdomains: it is not necessary to test any program from the set with two isomorphic inputs. Given a specific program under test and a specific property, we can merge several classes of nonisomorphic inputs into larger revealing subdomains, but without a priori knowledge of the program or the property, we cannot tell whether two nonisomorphic inputs will have the same behavior or not.

Isomorphism between inputs is the consequence of the semantics of a chosen set of programs and properties. Consider, for example, the set of all Java programs and properties that do not use the `System.identityHashCode` method, which returns a hash code based on the identity of an object. The behavior of any such program and property cannot depend on the actual identity of the objects. In Java, therefore, we define isomorphism with respect to object identity; each input structure is an object graph, and two inputs are isomorphic if they represent the same graph modulo a permutation of the objects.

Korat generates only nonisomorphic valid structures, namely, only nonisomorphic object graphs. When the valid structures that Korat generates constitute test inputs, the elimination of isomorphic structures significantly reduces the number of generated inputs without reducing the quality of the generated test suite. Sections 3.1.2 and 3.2.5 discuss isomorphism in detail.

1.2 Bounded-Exhaustive Testing

Bounded-exhaustive testing is a technique for testing the code on all inputs up to a given bound [56,72,104]. We call the numeric bound the *scope*, just like the numeric bound for Korat's solving. Korat can form the basis of a tool that performs either black-box or white-box bounded-exhaustive testing. In the black-box setting, the tool uses Korat to solve an imperative predicate that identifies the desired test inputs. Korat thus generates all test inputs, within a given scope, that satisfy the given properties; although Korat monitors the code of the predicate to generate the test inputs, it does not monitor the code under test in this setting. The tool then runs the code under test on each input and uses an oracle to check the correctness of each output. In the white-box setting, Korat additionally monitors the execution of the code under test, and it can establish the correctness for a set of inputs without necessarily running the code for all of them.

Bounded-exhaustive testing has detected faults in several real applications. We used a tool based on declarative predicates to detect faults in a naming architecture for dynamic networks and a constraint solver for first-order logic formulas [73]. Sullivan et al. [104] used the same tool to detect faults in a fault-tree analyzer; we recently used Korat for the same study and found an error in the declarative predicates that Sullivan et al. wrote. The Foundations of Software Engineering group at Microsoft Research implemented a solver for imperative predicates in the AsmL Test Tool (AsmLT) [33]. The use of AsmLT at Microsoft revealed faults in several systems, including web-service protocols and XML-processing tools [102].

1.2.1 Evaluation

Evaluations of techniques for test-input generation center around the quality of the generated test suites. But it is also important to consider the overall performance of testing, including generation of test inputs as well as checking of the output. In theory, bounded-exhaustive testing could detect any fault that exhaustive testing could detect: increasing the scope for the input size in the limit leads to exhaustive testing for all possible inputs. In practice, however, time constraints make it possible to test only up to certain scope, raising the possibility that the resulting nonexhaustive test suite may fail to detect a fault.

To evaluate the effectiveness of bounded-exhaustive testing, we used it to test a benchmark set of ten data-structure implementations. We developed five of these benchmarks following standard textbook algorithms [23], took four from the Java Collections Framework [105], and took one from the Intentional Naming System (INS) project [1]. Our experiments start from a correct version for each data structure. The original implementation of INS had faults [73], but we use a corrected version. During the development of the five benchmarks, the use of Korat helped us detect and correct several faults, adding anecdotal evidence of the usefulness of bounded-exhaustive testing.

The direct measure of the quality of a test suite is the number of detected faults. Thus, one approach to evaluating a technique for test-input generation is to apply it to several programs with known faults and measure how many faults the generated test suites detect. The problem with this approach is finding real programs with real faults that a technique can detect. Bounded-exhaustive evaluation has detected several faults in real programs, but our evaluation starts with corrected benchmarks. We thus must use another approach to evaluate our technique for test-input generation, namely, indirect metrics of the quality of test suites. In particular, we use (structural) *code coverage* and *mutation coverage.*

Code coverage is a common metric for comparing test suites [10]. Measuring code coverage requires executing the program on the inputs from a suite and recording the executed parts of the program. Statement coverage, for example, is the ratio of the number of executed statements to the total number of statements in the program; a similar ratio is defined for other parts, such as branches, conditions, or paths [10]. Our experiments use statement coverage and branch coverage, which we refer to as *code coverage.* Some parts may be infeasible—that is, impossible to execute for any input—so the denominator sometimes includes only feasible parts. Complete coverage is a ratio of 100%; in general it is undecidable whether a part is feasible or not, so we say that a test suite is *adequate* if it achieves a prespecified percentage. Code coverage is usually correlated with the ability of a test suite to detect faults: the higher the coverage that a test suite achieves, the better it is. According to this metric, a test suite that achieves complete code coverage is not worse than any other test suite.

Mutation coverage is another metric for comparing test suites [28, 44]. It determines how many seeded errors a test suite detects. A tool for measuring mutation coverage proceeds in two steps. The tool first produces a set of new, potentially faulty versions of a program, called *mutants*, by automatically performing syntactic modifications on the program, e.g., replacing a variable with another variable, say `n.left` with `n.right`. These modifications model typical errors committed by programmers. The tool then measures how many mutants a test suite detects: the tool runs the original program and each mutant on each input, and if a mutant generates an output different from the original program, the test input detects the error, and we say that it kills the mutant. Mutation coverage is the ratio of the number of killed mutants to the total number of mutants. The ideal ratio is again 100%, but some mutants may be semantically equivalent to the original program, so there is no input that can kill these mutants; we say that a test suite is *mutation-adequate* if it achieves a prespecified percentage. A test suite that detects a high percentage of seeded faults is likely to detect real faults [86]. Chapter 7 presents the mutation tool that we use in our experiments.

Chapter 8 presents the hypotheses that we evaluate and the details of the experiments for our set of benchmarks. The results show that bounded-exhaustive test suites can achieve high statement, branch, and mutation coverage for relatively small scopes. The results further show that Korat can generate inputs for these scopes in just a few minutes, often in a few seconds; additionally, checking the correctness of the code for all those inputs takes just a few minutes, often a few seconds. In summary, the results show that Korat is efficient enough for practical use in testing complex data structures.

We point out that it is not necessary to use code coverage and mutation coverage in conjunction with Korat. We used these two metrics to evaluate test suites that Korat generates. But users of Korat can instead use any other metric or simply use the scope itself and the time: generate test inputs and increase the scope until the testing becomes too slow. This approach is common in bounded checking of software artifacts [12, 51, 107].

1.3 Thesis and Challenges

Our thesis is that solving imperative predicates is a practical technique for testing software with structurally complex inputs: (1) it is feasible to generate, within small scopes, all valid nonisomorphic test inputs from imperative predicates that characterize structurally complex inputs, and (2) such test inputs are effective in software testing. This dissertation presents the evidence that supports our thesis.

We faced several challenges when we started this work:

- It might have been too difficult to write imperative predicates in practice. Our anecdotal experience shows that this is not the case. For example, two MIT undergraduate students wrote the Java predicates that check invariants for our data-structure benchmarks [72]. As another example, professional testers at Microsoft wrote predicates for testing several XML-processing tools, including an XPath compiler [102]. Both undergraduate students and professional testers were able to easily write imperative predicates in the languages they are familiar with.

- It might have been too difficult to write imperative predicates that can be efficiently solved. The efficiency of Korat's generation depends not only on the set of valid structuresbut also on the way the imperative predicate is written. Our experience shows that naturally written predicates induce good generation. The experimental results in Chapter 8 show the Korat's performance for predicates that we did not optimize for solving.

- It might have been too difficult to generate valid structures for imperative predicates that arise in practice. Developing a solver for imperative predicates was indeed our main challenge. Our previous experience [73] showed that it is feasible to generate some structurally complex test inputs using a specialized constraint language, but it was unclear whether we could do the same using a standard programming language. Moreover, the feasibility of generating complex data structures was questioned in the literature. The community was aware of the potential utility of generating structures that satisfy sophisticated invariants [5] but considered the problem to be too difficult to attempt to solve using random generation:

  > Trying to generate a random heap state, with a rats' nest of references, and then select those that represent queues, would be both difficult and hopeless in practice. [20, page 68]

  We agree that generating complex structures by generating random graphs and filtering out those that do not satisfy the properties of interest is hopeless, albeit easy to implement. Our results show, however, that it is feasible to efficiently generate structures using bounded-exhaustive solving of imperative predicates. This is feasible for structures that satisfy not only the queue property but many other data-structure invariants as specified by arbitrary pieces of code written in a standard programming language.

- The generation might have been too slow to be practical, or the generated test suites might have been of low quality. (As discussed in Section 1.2.1, it is necessary to consider the performance of the generation and the quality of the test suites jointly in bounded-exhaustive testing.) Our results in Chapter 8 again show that this is not the case: the generation is practical, and the test suites achieve high coverage according to two metrics. Together with other studies that used bounded-exhaustive testing to detect faults in real applications [56, 73, 102, 104], these results suggest that systematic generation within a small scope [13, 33, 108] is a practical way to obtain test suites for code that manipulates complex data structures.

1.4 Contributions

This dissertation makes the following contributions:

- **Imperative predicates as constraints:** We propose imperative predicates as a practical means of specifying properties of the desired test inputs. Whereas previous research used special-purpose languages for specifying constraints on test inputs, we allow the use of standard programming languages such as Java. We formalize the problem of solving imperative predicates as finding, within given bounds, all nonisomorphic inputs, called structures, for which the predicate returns true.

- **Solver for imperative predicates:** We present a technique for solving imperative predicates and a solver for Java predicates, Korat. Given a predicate and a finitization that bounds the structures that are inputs to the predicate, Korat systematically searches the structure space to generate all valid, nonisomorphic structures. Korat dynamically monitors the executions of the predicate to determine the relevant fields of the structure, i.e., the fields that affect the result of predicate's computation. Korat then eliminates from consideration all additional structures with identical values for relevant fields.

- **Correctness of the solver:** We formalize the Korat algorithm and present a proof of its correctness. The correctness of Korat is important for evaluating Korat accurately. The correctness is also important in practice—if Korat mistakenly produces a structure that does not satisfy the desired property, the developer may waste valuable time attempting to track down a nonexistent error in the implementation. If, on the other hand, Korat incorrectly causes a set of structures to be systematically omitted, the generated test inputs will not detect any fault that is triggered only by those omitted inputs.

- **Extension to the solver:** We present *dedicated generators*, extensions to Korat that show how to exploit the presence of common input properties to further prune the search during generation. Dedicated generators increase Korat's effectiveness in bounded-exhaustive testing. They optimize the generation for common properties that often appear within test inputs, without eliminating any valid structures. Moreover, dedicated generators provide a library of imperative predicates that can be reused when developing new imperative predicates.

- **Implementation of a testing tool set:** We describe the design and implementation of three tools. First, we present details of our Korat implementation for Java and discuss some possible alternative implementations. Second, we present our testing tool built on top of Korat. Third, we present a tool for mutation testing of Java programs, which we developed to evaluate bounded-exhaustive testing.

- **Applications in testing:** We describe how to apply the solving of imperative predicates in testing. Our initial application was in black-box, specification-based unit testing. Given a formal specification for a method, a testing tool synthesizes an imperative predicate from the method precondition. It then uses a solver to generate all nonisomorphic test inputs within a scope, executes the method on each test input, and uses the method postcondition as a test oracle to check the correctness of each output. We additionally present how to apply the solving of imperative predicates in white-box testing, testing sequences of method calls, and system testing.

- **Evaluation:** We present experimental results that show the feasibility of testing data structures using solvers for imperative predicates, Korat in particular. Specifically, our results show that (1) bounded-exhaustive testing within small scopes can generate test suites that are mutation-adequate; (2) bounded-exhaustive test suites can achieve complete code coverage for even smaller scopes, but such suites do not detect all mutants; (3) it is practical to use Korat to generate inputs and check correctness for these scopes; and (4) bounded-exhaustive testing within a scope is often more effective than testing with the same number of randomly selected inputs from a larger scope.

1.5 Organization

The remainder of this dissertation is organized as follows. Chapter 2 illustrates with two examples how programmers can use Korat to generate inputs and then test their programs. Chapter 3 formalizes the problem of solving imperative predicates. This chapter also presents details of Korat. Chapter 4 shows a proof of correctness for Korat. Chapter 5 discusses dedicated generators, an extension to Korat. Chapter 6 then presents applications of Korat in testing, with emphasis on bounded-exhaustive testing. Chapter 7 presents the design and implementation of our tool for mutation testing of Java programs. Chapter 8 presents an evaluation of Korat and bounded-exhaustive testing. Chapter 9 compares testing based on imperative predicates and Korat with our previous approach for testing based on declarative predicates. Chapter 10 discusses other related work. Chapter 11 discusses the strengths and limitations of Korat and concludes this dissertation.

Chapter 2

Examples

This chapter presents two examples that illustrate how programmers can use Korat to test their programs. The first example uses a binary tree data structure to illustrate testing a method that manipulates a linked data structure. The second example uses a heap data structure to illustrate testing a method that manipulates an array-based data structure. The term *heap* here refers to the data structure, also known as priority queues, and not to the dynamically-allocated memory. We use the term *object graph* to refer to the latter.

2.1 Binary Search Tree

We use a method that removes an element from a binary tree to illustrate generation and testing of linked data structures. Although removal is conceptually simple, the implementation involves intricate details to restore a tree after removing an inner node. Figure 2-1 shows a part of Java code for a binary search tree implementation of a set of integers. Each object of the class `SearchTree` represents a binary search tree. The value of the `size` field is the number of nodes in the tree. Objects of the inner class `Node` represent nodes of the trees. The tree stores the elements of the set in the `info` fields. The `remove` method removes an element from a tree. (Appendix A shows the full code for an example `remove` method.)

Figure 2-1 also shows the informal specification for `remove`: this method removes the element if it was in the tree, and returns `true` iff it removes the element. Each input to the `remove` method conceptually consists of a tree and an element to remove. More precisely, the input consists of a `SearchTree` pointer, which is the value of the implicit argument `this`, and an element to remove, which is the value of the argument `info`. The pointer `this` is the root of an object graph that represents the state of the tree right before the execution of `remove`, i.e., in the *pre-state.* The execution can modify this object graph. Each output thus consists of a potentially modified tree and a boolean that indicates whether the execution has removed the element; the pointer `this` is the root of a modified object graph that represents the state of the tree right after the execution of `remove`, i.e., in the *post-state.*

To test the `remove` method, the programmer needs to (1) generate several inputs

```
class SearchTree {
    int size;  // number of nodes in the tree
    Node root; // root node
    static class Node {
        Node left;  // left child
        Node right; // right child
        int info;   // data
    }

    /** Removes an element from the tree.
        Requires that "this" be a binary search tree.
        Ensures that "info" is removed, if it was in the tree.
        Returns "true" if the element was removed, "false" otherwise. */
    boolean remove(int info) {
        ... // for method body see Appendix A
    }

    boolean repOk() { // for helper methods see Appendix A
        // checks that empty tree has size zero
        if (root == null) return size == 0;
        // checks that the object graph is a tree
        if (!isTree()) return false;
        // checks that size is consistent
        if (numNodes(root) != size) return false;
        // checks that data is ordered
        if (!isOrdered(root)) return false;
        return true;
    }
}
```

Figure 2-1: Binary search tree example.

for `remove`, (2) execute `remove` for each input, and (3) check the correctness of each output. We focus here on automating test-input generation using Korat; we show later how to automate execution and checking (Section 6.1.2). To use Korat, the programmer needs to provide an imperative predicate that checks the validity of inputs. Following Liskov [70], we call such predicates `repOk`; they typically check the *representation invariant* (or class invariant) of the corresponding data structure. Good programming practice suggests that implementations of abstract data types always provide such predicates, because they are useful for checking partial correctness[1] of the implementations [70]. When programmers follow this practice, the predicate is already present, and Korat can generate test inputs almost for free.

For `remove`, a valid input should be an object graph that is actually a binary search tree, not an arbitrary graph. Figure 2-1 shows the `repOk` method for `SearchTree`; `repOk` is a Java predicate that checks that the input is indeed a valid

[1] A `repOk` predicate can check whether a method preserves the class invariant. To check a stronger property for a specific method, the programmer needs to provide, in addition to `repOk`, a predicate that checks this stronger property.

binary search tree with the correct `size`. First, `repOk` checks if the tree is empty. If not, `repOk` checks that there are no undirected cycles in the object graph reachable from the `root` field along the `left` and `right` fields. It then checks that the number of nodes reachable from `root` is the same as the value of the field `size`. It finally checks that all elements in the left/right subtree of a node are smaller/larger than the element in that node. (Appendix A shows the full code for `repOk` and the methods it invokes.)

It is worth pointing out that the same `repOk` predicate also checks partial correctness for all other methods in `SearchTree`, e.g., `add`. Manually developing a test suite that achieve high code coverage for all methods in a data structure can often be much harder than writing a `repOk` invariant from which Korat automatically generates test inputs that can achieve high code coverage. As a simple illustration, a test suite that would achieve complete branch coverage for `remove` and `add` would have more lines of code than the `repOk` predicate.

The programmer can use Korat to generate valid test inputs for the `remove` method by generating valid structures for the `repOk` predicate. Each input is a pair of a tree and an element, where the tree satisfies `repOk`, and the element is unconstrained. To limit the number of inputs, the programmer provides Korat with a *finitization* (Section 3.2.2) that specifies bounds on the number of objects in the data structures and on the values in the fields of these objects. For trees, the finitization specifies the maximum number of nodes and the possible elements; a tree is in a scope s if it has at most s nodes and s elements. We can use the integers from 1 to s for the tree elements and for the `info` argument.

Given a finitization and a value for scope, Korat automatically generates all valid structures for the predicate and thus test inputs for the method. More precisely, Korat generates only the structures that are *nonisomorphic*. In our running example, two binary search trees are isomorphic if they have the same underlying graph and elements, irrespective of the identity of the actual nodes in the trees. Section 3.1.2 gives a precise definition of isomorphism; it suffices to say that nonisomorphism can significantly reduce the number of test inputs without reducing the quality of the test suite. For example, in scope three, Korat generates 45 nonisomorphic test inputs. Figure 2-2 shows the 15 nonisomorphic trees of size up to three; each input is a pair consisting of one of these 15 trees and one of the three values for `info`.

2.1.1 Preview of the Experimental Results

We present here a sample of experimental results for `SearchTree`; Chapter 3 describes how Korat generates valid structures, and Chapter 8 presents a detailed evaluation of Korat and its use in testing. We discuss the efficiency of generating test inputs for `SearchTree`, the efficiency of checking correctness for those inputs, and the quality of those inputs. The experiments evaluate Korat for both `remove` and `add` methods for `SearchTree`. The inputs for these methods are the same, although their behavior differs.

In scope three, for example, Korat generates 90 test inputs for both `remove` and `add` methods (45 inputs each) in less than a second. As another example, in scope

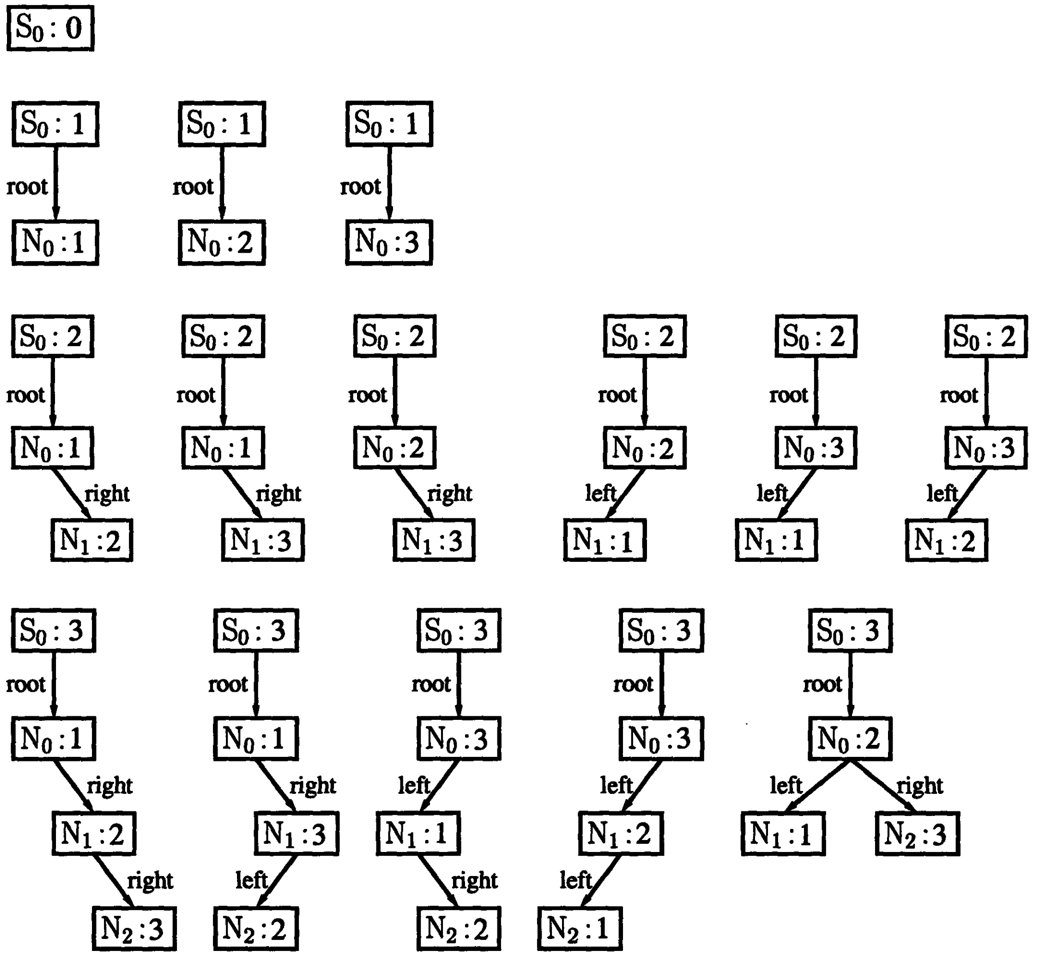

Figure 2-2: Trees that Korat generates for scope three. Each tree consists of one `SearchTree` object (S_0) and several `Node` objects (N_0, N_1, and N_2). For each object, we show the identity and the value of the primitive field (`size` for S_0 and `info` for the `Node` objects). Edges represent values of reference fields with no edge for `null`.

seven, Korat generates 41,300 inputs for `remove` and `add` in less than ten seconds. With dedicated generators (Section 5.3), the time that Korat takes to generate these 41,300 inputs further reduces to less than three seconds. We discuss below that the test suite that Korat generates for scope seven is both code-coverage adequate and mutation adequate, which means that the test suite has a high quality according to these two metrics. Yet it takes Korat only a few seconds to generate this test suite.

We can check the code under test for the inputs that Korat generates using an *oracle* that automatically determines whether the code behaves correctly for each given input. Chapter 6 describes how to obtain such oracles by translating specifications, such as those written in the Java Modeling Language [17,67], into run-time assertions. After generating the inputs with Korat, our testing tool invokes the method under test on each input and reports a counterexample if the method violates an assertion.

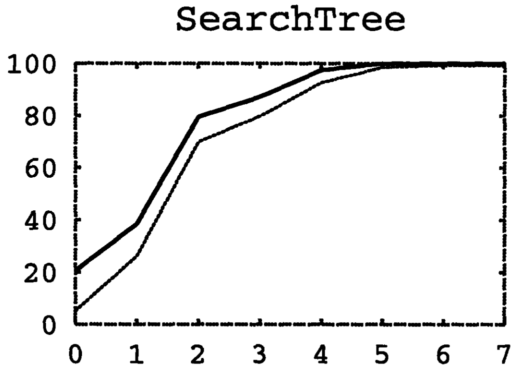

Figure 2-3: Variation of code coverage and mutating coverage with scope. Thick line shows the statement code coverage and thin line the mutation coverage.

This process checks the correctness of the method for the given scope. In scope three, for example, it takes less than a second to check both `remove` and `add` for all 90 inputs, and in scope seven, it takes less two seconds to check `remove` and `add` for all 41,300 inputs.

Korat efficiently generates inputs that enable testing of `remove` and `add` methods for small scopes such as three and seven, but the question is how good are these inputs? Our implementation of `SearchTree` is correct, so Korat cannot detect a fault; instead, we use indirect metrics to estimate the quality of the test suites, in particular code coverage and mutation coverage. Figure 2-3 shows how code coverage and mutation coverage vary with the scope for `SearchTree`. We can see that scope five is sufficient to achieve complete coverage for this benchmark. Hence, the test inputs that Korat generates for scope five are adequate ("good enough") according to a testing criterion based on code coverage [10]. However, we can see (from Figure A-3) that killing all nonequivalent mutants requires inputs from scope six. Mutation criterion is a stronger test criterion than code-coverage criterion: in this example, Korat needs a larger scope to generate inputs that are mutation adequate [28]. But with respect to both of these criteria, Korat has excellent performance. For both scopes five and six, generating inputs and checking correctness takes less than 15 seconds. It is thus practical to use Korat to generate test inputs for `SearchTree` methods.

2.2 Heap Array

We use a method that removes the largest element from a heap data structure [23] to illustrate generation and testing of array-based data structures. The (binary) heap data structure can be viewed as a complete binary tree—the tree is completely filled on all levels except possibly the lowest, which is filled from the left up to some point. Heaps also satisfy the heap property—for every node *n* other than the root, the value of *n*'s parent is greater than or equal to the value of *n*.

```
class HeapArray {
    int size; // number of elements in the heap
    Comparable[] array; // heap elements

    /** Removes the largest element from the heap.
        If the heap is nonempty, returns the largest element.
        If the heap is empty, throws an exception. */
    Comparable extractMax() {
        // ... method body
    }

    boolean repOk() {
        // checks that array is non-null
        if (array == null) return false;
        // checks that size is within array bounds
        if (size < 0 || size > array.length)
            return false;
        for (int i = 0; i < size; i++) {
            // checks that elements are non-null
            if (array[i] == null) return false;
            // checks that array is heapified
            if (i > 0 && array[i].compareTo(array[(i-1)/2]) > 0)
                return false;
        }
        // checks that non-heap elements are null; no leaks
        for (int i = size; i < array.length; i++)
            if (array[i] != null) return false;
        return true;
    }
}
```

Figure 2-4: Heap array example.

Figure 2-4 shows Java code that declares an array-based heap and defines the corresponding `repOk` predicate that checks whether the input is a valid `HeapArray`. The `array` contains the elements of the heap, which implement the interface `Comparable` that provides the method `compareTo` for comparisons. The `repOk` predicate first checks for the special case when `array` is `null`. If it is not, `repOk` checks that the `size` of the heap is within the bounds of the `array`. Then, `repOk` checks that the array elements that belong to the heap are not `null` and satisfy the heap property. Finally, `repOk` checks that the array elements that do not belong to the heap are `null`.

Figure 2-4 also shows the declaration of `extractMax` method that removes and returns the largest element from the heap. This method has two different behaviors with actual preconditions: the precondition for the normal behavior requires the input to be nonempty, and the precondition for the exceptional behavior requires the input to be empty. To use Korat to generate inputs that test only the normal behavior, the programmer needs to provide an imperative predicate that checks whether the implicit `this` argument for `extractMax` satisfies `this.repOk() && this.size >`

```
size = 0, array = []
size = 0, array = [null]
size = 1, array = [Integer(0)]
size = 1, array = [Integer(1)]
```

Figure 2-5: Heaps that Korat generates for scope one.

0; similarly, for the exceptional behavior, the predicate should be `this.repOk() && this.size == 0`. To generate inputs for both behaviors, the predicate should be the disjunction of the predicates for the two behaviors. Section 6.1.2 shows how to automatically generate such predicates from imperative preconditions.

Korat generates heaps for a given finitization that specifies a bound for the heap `size`, a bound for the length of the `array`, and the values for the array elements. The elements need to implement the interface `Comparable`, and they need to be concrete objects that Korat can create. For example, the elements may be from a set of objects of the class `Integer`, and the finitization specifies that each array element either points to one of those objects or is `null`. A heap is in a scope *s* if the size, array length, and array elements range from 0 to *s*.

Given a finitization and a value for scope, Korat automatically generates all heaps. For example, Figure 2-5 shows the four heaps that Korat generates for scope one. These heaps are for testing both behaviors of `extractMax`: the top two heaps are empty and test the exceptional behavior, whereas the bottom two heaps are nonempty and test the normal behavior. Note that both empty heaps have the value of the `size` zero but differ in the array length. Korat generates both of these heaps because they are not isomorphic, and `extractMax` could have different behavior for them; for a correct `extractMax`, these two heaps are equivalent, and `extractMax` has the same behavior for both—throws an exception. As another example, in less than one second, Korat generates 1,919 heaps for scope five.

Our experiments use Korat to generate inputs for and test methods `extractMax` and `insert` (which inserts an element in the heap). It takes less than three seconds to generate 118,251 inputs and check both methods for scope six that achieves complete code coverage. It takes less than 25 seconds to generate 1,175,620 inputs and check both methods for scope seven that kills all nonequivalent mutants. These results show that it is practical to use Korat to thoroughly test `HeapArray` methods. In under half a minute, it is possible to test these methods for all inputs in scope seven, and these inputs are coverage adequate and mutation adequate, i.e., have a high quality according to those two criteria.

Chapter 3

Solving Imperative Predicates

This chapter formally presents imperative predicates, solving imperative predicates in general, and Korat, our solver for imperative predicates. An imperative predicate is a piece of code that takes an input, which we call structure, and returns a boolean; solving an imperative predicate amounts to finding those structures for which the predicate returns true. This chapter presents in detail how Korat solves imperative predicates and gives a pseudo-code algorithm of Korat; Chapter 4 proves the correctness of this algorithm.

3.1 Problem Statement

We start by defining imperative predicates and their inputs. This allows us to precisely state the problem of solving imperative predicates and its correctness requirement. We then compare solving imperative predicates with the satisfiability problem in logic. We next present a naive algorithm for solving imperative predicates and discuss why it is impractical. We finally present how to reduce a predicate with several inputs to a predicate with one input.

3.1.1 Definitions

We first define imperative predicates. We then define input structures and validity and isomorphism of structures.

Definition 1 (Imperative Predicate). *An* imperative predicate *is a piece of imperative code that takes an input* structure *and returns a boolean.* □

For example, Java programs represent the state with an object graph, and a Java predicate is a method that takes a pointer, which is a root of an object graph, and returns a boolean. In the narrow sense, an input to a predicate is just a pointer, but in the broad sense, an input is the whole object graph reachable from that pointer. Recall from Section 2.1 the `repOk` predicate that checks whether its input satisfies the invariant for a binary search tree. The input to `repOk` is effectively the whole object graph reachable from the `SearchTree` pointer that is the value of the implicit

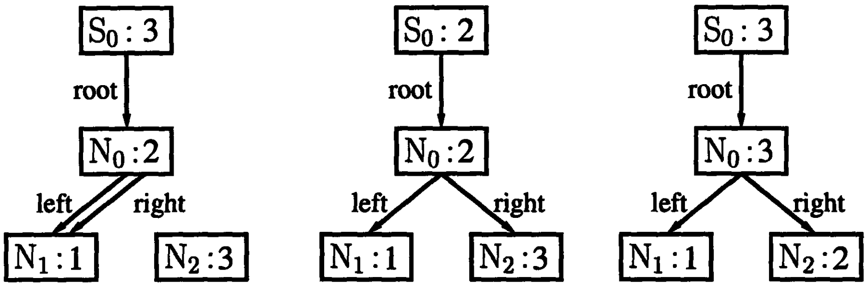

Figure 3-1: Example object graphs that are not binary search trees. The leftmost graph is not a tree, the middle graph has an incorrect number of nodes, and the rightmost graph does not have values in a binary search arrangement.

this argument. If root is not null, this object graph contains at least one Node object.

There is some natural space of input structures for each predicate. For Java predicates, the space consists of all object graphs that are well-typed according to the Java type rules [38]. Since Java is a type-safe language, we do not need to consider object graphs that are not well-typed, e.g., where pointers would have arbitrary (integer) values. Instead, pointers point to actual objects of the appropriate types; for instance, SearchTree pointers point to SearchTree objects, and Node pointers point to Node objects. In languages, such as C or C++, that are not type-safe, pointers can have arbitrary values, but we still consider only the structures that satisfy Java-like type rules.

We define structures as rooted (object) graphs, where nodes represent objects and edges represent fields. Let O be a set of objects whose fields form a set F. Each object has a field that represents its class. Each array object has fields that are labeled with array indexes and point to array elements. Let P be the set consisting of null and all values of primitive types, such as int.

Definition 2 (Structure). *A* structure *is a rooted, edge-labeled graph $\langle O, r, E\rangle$, where $r \in O$ is an object called root, and $E \subseteq O \times F \times O \cup P$ such that for every object $o \in O$ and every field $f \in fields(o)$, there is exactly one $\langle o, f, o'\rangle \in E$ (where o' has the appropriate type according to the type of f).*

We write reachable($\langle O, r, E\rangle$) for a set of objects reachable from the root r. Formally, reachable($\langle O, r, E\rangle$) $= O' \subseteq O$ is the smallest set O' that satisfies these two equations: $r \in O'$ and $o \in O' \Rightarrow \{o' \in O | \exists f. \langle o, f, o'\rangle \in E\} \subseteq O'$.

We next consider validity of input structures. Our example repOk predicate returns true or false for each well-typed structure (with a SearchTree root). Figure 2-2 (Section 2.1) shows some structures for which repOk returns true. There are (many more) other structures for which repOk returns false. Figure 3-1 shows some examples of such structures: the first structure has an underlying graph that is not a tree as there is sharing between nodes; the second structure has an incorrect

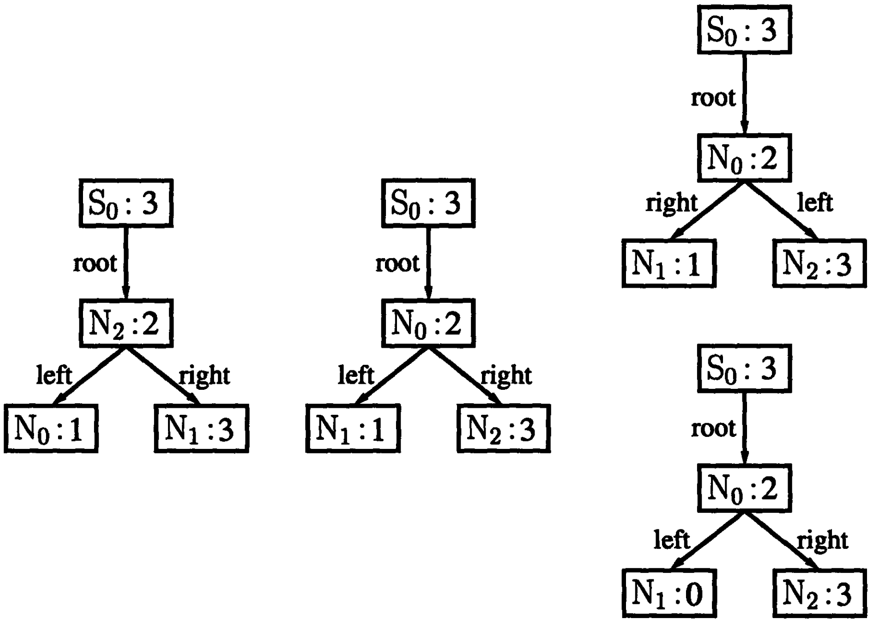

Figure 3-2: Example (non)isomorphic object graphs. The graph in the middle is isomorphic to the graph on the left (only the identity of the nodes differ) but not isomorphic to any of the graphs on the right (the edges or primitive values differ).

size as there are three, not two, nodes reachable from root; and the third structure has elements that do not satisfy the binary search property as the element two is in the right subtree of the element three.

Most practical predicates are deterministic, i.e., given the same input structure, any two executions of the predicate are intuitively the same and produce identical result. In general, however, predicates may be nondeterministic either only in the execution (i.e., given the same structure, two executions may differ but produce identical result) or even in the result (i.e., given the same structure, two executions may produce different results). Each execution of a predicate produces a specific result (true or false), but for nondeterministic predicates some executions may produce true, and some executions may produce false, for the same structure.

Definition 3 (Validity). *Let γ be a structure for a predicate π, and let $\pi(\gamma)$ be the set of results that executions of π can produce for structure γ. We say that γ is* valid *iff $\pi(\gamma)=$ {true}, and γ is* invalid *iff $\pi(\gamma)=$ {false}. We also say that a valid structure* satisfies *the predicate.* □

Note that a structure may be neither valid nor invalid; more precisely, a structure γ is neither valid nor invalid when $\pi(\gamma)=$ {false,true}.

We define isomorphism between input structures based on object identity; two structures are isomorphic iff the object graphs reachable from the root are isomorphic. Figure 3-2 shows some example (non)isomorphic structures.

Definition 4 (Isomorphism). *Let $O_1, \ldots, O_n$ be some sets of objects from n classes. Let $O = O_1 \cup \ldots \cup O_n$, and suppose that structures consist only of objects from O and primitive values from P. Two structures, $\langle O, r, E\rangle$ and $\langle O, r', E'\rangle$, are* isomorphic *iff there exists a permutation p on $O \cup P$ that is identity on P and that maps objects from O_i to objects from O_i for all $1 \leq i \leq n$, such that $p(r) = r'$ and*

$$\forall o \in \mathit{reachable}(\langle O, r, E\rangle).\ \forall f \in \mathit{fields}(o).\ \forall v \in O \cup P.$$
$$\langle o, f, v\rangle \in E \Leftrightarrow \langle p(o), f, p(v)\rangle \in E'.$$

□

The definition corresponds to so-called *unlabeled graphs* in combinatorics [101]. Note that it allows only the identity of the objects to permute between isomorphic structures; the edge labels (fields and array indexes) and primitive values cannot permute. Note also that this definition relies on there be exactly one $\langle o, f, v\rangle$ for each field f of an object o. This is indeed the case for object graphs and structures (Definition 2), but not for general graphs. We discuss this further in Section 9.2.1.

3.1.2 Formal Correctness Statement

We next precisely state the correctness requirement for solving imperative predicates. It requires a solver for imperative predicate to find valid structures for a given predicate. For our example `repOk` predicate, as well as in most practical situations, the space of all input structures (and the number of valid structures) are infinite. Specifically, the type for `Node` is recursive, and it is thus possible to build object graphs of arbitrary size; also, the value of the field `size` is unbounded in theory[1]. We consider only a finite portion of the structure space: the user provides bounds that limit the size of the structure and the values that primitive fields can take.

We require a solver for imperative predicates to generate *all* nonisomorphic valid structures within specified bounds. Isomorphism between structures partitions the structure space into *isomorphism partitions*, and we require the solver to generate one representative from each isomorphism partition of valid structures. More precisely, given a predicate, the solver generates a set of structures with these properties:

- **Soundness:** The solver generates no invalid structure, i.e., no set contains a structure for which the predicate always returns `false`.

- **Completeness:** The solver generates at least one valid structure from each isomorphism partition of valid structures, i.e., every set contains at least one representative from each isomorphism partition of structures for which the predicate always returns `true`.

- **Isomorph-Freeness:** The solver generates at most one structure from each isomorphism partition, i.e., no set contains two isomorphic structures.

From the last two properties, we conclude that the solver should generate exactly one valid structure from each isomorphism partition that contains valid structures. Note

[1] In Java, the semantics specifies that primitive values of type `int` have exactly 32 bits.

```
boolean areIsomorphic(Object o1, Object o2) {
    return areIsomorphic(o1, o2,
        new Map<Object, Object>(), new Map<Object, Object>());
}
boolean areIsomorphic(Object o1, Object o2, Map m12, Map m21) {
    if ((o1 == null) || (o2 == null)) return (o1 == o2);
    if (m12.containsKey(o1)) return (m12.get(o1) == o2);
    if (m21.containsKey(o2)) return (m21.get(o2) == o1);
    if (o1.getClass() != o2.getClass()) return false;
    m12.put(o1, o2);
    m21.put(o2, o1);
    foreach (Field f in o1.getFields()) {
        if (f.isPrimitiveType()) {
            if (o1.f != o2.f) return false;
        } else {
            if (!areIsomorphic(o1.f, o2.f, m12, m21)) return false;
        }
    }
    return true;
}
```

Figure 3-3: Algorithm for checking isomorphism of two object graphs.

also that the solver should not generate any invalid structure, but it may or may not generate structures that are neither valid nor invalid.

Since structures (and thus valid structures) are rooted and deterministically edge-labeled graphs, it is easy to check isomorphism; Figure 3-3 shows an algorithm that checks isomorphism of two (potentially cyclic) structures in $O(n + e)$ steps, where n is the number of objects in the structures, and e is the total number of fields. One approach for generating only nonisomorphic valid structures would be to generate all valid structures and then filter out the isomorphic ones. However, we show later (Section 3.2.5) that our solver, Korat, does not require any such filtering.

3.1.3 Naive Algorithm

After the user limits the size of the structures, the structure space for the predicate is finite. Suppose that we can enumerate all structures from the structure space, regardless of their validity. (Section 3.2.3 presents how to do that.) We can consider the following algorithm for generating valid structures:

1. Enumerate each structure from the structure space.

2. Execute the predicate for that structure.

3. If the predicate returns `true`, print the structure.

This algorithm clearly generates all valid structures. (To additionally generate only nonisomorphic structures, the algorithm would need to have a post-processing step that filters out isomorphic valid structures.)

```
class SearchTree {
    boolean contains(int info) {
        ...
    }
}

class TreeInfoPair {
    SearchTree This;
    int info;
    boolean repOk() {
        return This.contains(info);
    }
}
```

Figure 3-4: Example of a predicate with multiple inputs. We can automatically reduce the predicate to a predicate with one input.

The problem with this algorithm is that it enumerates the *entire* structure space to find the valid structures. However, most structure spaces, including the structure space for `repOk` for `SearchTree`, are *sparse*: although there is a very large number of structures, only a tiny fraction of them are valid. Section 8.2 shows the results that support this statement; for example, for `SearchTree` of size seven, there are 2^{59} structures in the structure space, and only $7! \cdot 429$ of them are valid, i.e., only one in 10^{11} structures is valid. (Additionally, each valid structure belongs to an isomorphism partition with 7! structures, so there are only 429 valid nonisomorphic structures.) In the limit, as the size of the structures tends to infinity, the ratio of valid structures and all structures typically tends to zero; as mentioned, however, we only consider a finite portion of the structure space. To have a practical algorithm for solving imperative predicates, we need to be able to better handle sparse structure spaces.

3.1.4 Predicates with Multiple Inputs

So far we have considered imperative predicates with only one input pointer, which directly corresponds to structures with one root. We next show how to reduce a predicate with multiple inputs, either pointers or primitive values, to a predicate with one input pointer. In general we do this by: (1) introducing a class whose objects represent tuples with multiple inputs and (2) replacing the multiple inputs with one input of the new class. We next illustrate this on an example.

Figure 3-4 shows a predicate `contains` that takes a tree (implicit argument `this`) and an element and checks whether the element is in the tree. This predicate conceptually has two inputs, so we introduce a new class, say `TreeInfoPair`, that represents a pair of a tree and an element. We then translate the predicate `contains` to a predicate on the objects of the new class; as usually, we call this new predicate `repOk`. We can now use a solver to generate valid structures for `repOk`; each structure represents a pair of inputs for which `contains` returns `true`.

```
void koratSearch(Predicate pred, Finitization fin) {
    initialize(fin);
    while (hasNextCandidate()) {
        Object candidate = nextCandidate();
        try {
            if (pred(candidate))
                output(candidate);
        } catch (Throwable t) {}
        backtrack();
    }
}
```

Figure 3-5: High-level overview of the Korat search algorithm

3.2 Korat: A Solver for Imperative Predicates

We next describe our technique for solving imperative predicates and a tool, called Korat, that implements this technique for Java predicates. Although our presentation is in terms of Java, the technique generalizes to other languages. We first give a high-level overview of the Korat algorithm. Korat uses two optimizations—pruning based on accessed fields and elimination of isomorphic structures—to improve on the naive search, without comprising the completeness. We then describe parts of the Korat algorithm in detail. We use the `repOk` method from `SearchTree` as an example predicate and illustrate how Korat generates valid binary-search trees. (Section 6.1.2 presents how Korat generates valid test inputs for the `remove` method.) We finally present details of the implementation.

3.2.1 Overview

Given a Java predicate and a bound on its input structures, Korat automatically generates all nonisomorphic structures that are *valid*, i.e., structures for which the predicate always returns `true`. Korat uses a *finitization* (Section 3.2.2) to bound the size of the input structures to the predicate. To enable efficient search for valid structures, Korat encodes each structure with a *state*, and each finitization bounds the *state space* (Section 3.2.3). Korat uses backtracking (Section 3.2.4) to systematically search this state space to find all valid structures.

Figure 3-5 shows a high-level overview of the Korat search algorithm. Based on a finitization for the input structures to the predicate, Korat first initializes the state space of structures. Korat then builds *candidate* structures and executes the predicate on them to check their validity. Korat monitors these executions to dynamically determine which parts of the candidate the result of the predicate depends on. More specifically, Korat monitors the fields of the candidate that the execution accesses. If the predicate returns `true`, Korat outputs the candidate. Otherwise, if the predicate returns `false` or throws an exception, Korat skips the candidate. Korat then backtracks on the values of the accessed fields to generate the next candidate. Korat terminates when it explores the entire state space.

```
Finitization finSearchTree(int numNode,
        int minSize, int maxSize, int minInfo, int maxInfo) {
    Finitization f = new Finitization(SearchTree.class);
    ObjSet nodes = f.createObjects("Node", numNode); // #Node = numNode
    nodes.add(null);
    f.set("root", nodes); // root in null + Node
    // size in [minSize..maxSize]
    f.set("size", new IntSet(minSize, maxSize));
    f.set("Node.left", nodes); // Node.left in null + Node
    f.set("Node.right", nodes); // Node.right in null + Node
    // Node.info in [minInfo..maxInfo]
    f.set("Node.info", new IntSet(minInfo, maxInfo));
    return f;
}

Finitization finSearchTree(int scope) {
    return finSearchTree(scope, 0, scope, 1, scope);
}
```

Figure 3-6: Two finitizations for the `repOk` method.

As mentioned, naive checking of all possible candidate structures would prohibit searching sparse state spaces that arise in practice. Korat uses two optimizations. First, Korat prunes the search based on the accesses that the executions of the predicate make to the fields of the candidate structures. To monitor the accesses, Korat instruments the predicate and all methods that the predicate transitively invokes (Section 3.2.6). Intuitively, this pruning is correct because validity of the candidate must be independent of the values of the fields that the predicate does not read before returning the result. Second, Korat generates only nonisomorphic candidates and thus only nonisomorphic valid structures (Section 3.2.5). These two optimizations speed up the search without compromising the completeness. The results in Section 8.2 show that the number of candidates that Korat explores is only a tiny fraction of the whole structure space. For example, for `SearchTree` of size seven, Korat explores less than 350,000 candidates out of 2^{59} possible structures in the structure space, i.e., less than one candidate in 10^{12} structures.

3.2.2 Finitization

Korat uses a finitization, i.e., a set of bounds that limits the size of the structures, to generate a finite state space for predicate's structures. Each finitization provides (1) limits on the number of objects of each class and (2) the values for each field of those objects. The structures can consist of objects from several classes, and the finitization specifies the number of objects for each of those classes. A set of objects from one class forms a *class domain.* The finitization also specifies a set of values for each field; this set forms a *field domain*, which is a union of several class domains and constants. The constants include `null` for pointers and all primitive values for fields of primitive types.

```
class ClassDomain { // ordered class domain
    Object[] objects;
}
class ClassDomainIndex {
    ClassDomain domain;
    int index; // index into 'domain.objects' array
}
class ObjField { // field of an object from some domain
    Object object;
    Field field;
}
```

Figure 3-7: Classes that Korat uses to represent state space.

We need an explicit programming model for writing finitizations. In the spirit of Extreme Programming [8] that uses the implementation language familiar to programmers for writing tests and partial specifications, Korat provides a library that allows the user to write finitizations in Java. The initial version of Korat provided a special-purpose language that allows writing finitizations more succinctly than in Java. (This language is sketched in the comments in Figure 3-6.) However, adding a new language would put additional burden on the user.

Figure 3-6 shows two methods that create finitizations for the example `repOk`; each finitization builds an object of the `Finitization` class from the Korat library. Invoking `finSearchTree(scope)` creates a finitization for scope `scope`. The `createObjects` method specifies that the structure contains at most `numNode` objects from the class `Node`. The `set` method specifies a field domain for each field: the fields `root`, `left`, and `right` can point to either `null` or a `Node` object; the `size` field ranges between `minSize` and `maxSize`; the `info` field ranges between `minInfo` and `maxInfo` (specified using the utility class `IntSet`). The Korat library provides several additional classes for easy construction of class domains and field domains.

Korat automatically generates a *skeleton* for finitization methods from the type declarations in the Java code. Actually, the first finitization method in Figure 3-6 is generated by Korat. The automatic generation of skeletons does not always produce a method to the user's liking. The user can thus further specialize or generalize the skeleton, as the second finitization method in Figure 3-6 shows. Note that programmers can use the full expressive power of the Java language for writing finitization methods. The AsmLT tool [33] that implements a solver for predicates written in the AsmL language additionally provides a graphical user interface for specifying finitizations.

It is up to the user to choose the values for field domains in the finitizations. The guideline for writing successful finitizations is to provide the necessary values such that it is possible to generate the valid structures, and sufficient values, such that the state space is not larger than necessary to generate valid structures. For example, it is necessary to provide the value `null` for the fields `left` and `right`: each leaf in a (nonempty) valid tree has this value. Also, it is necessary to provide for `info` at least

S_0		N_0			N_1			N_2		
root	size	left	right	info	left	right	info	left	right	info
□	□	□	□	□	□	□	□	□	□	□
null	0	null	null	1	null	null	1	null	null	1
N_0	1	N_0	N_0	2	N_0	N_0	2	N_0	N_0	2
N_1	2	N_1	N_1	3	N_1	N_1	3	N_1	N_1	3
N_2	3	N_2	N_2		N_2	N_2		N_2	N_2	

Figure 3-8: State space for `finSearchTree(3)`.

as many values as there are nodes: a valid tree has no repeated elements in the nodes. However, to generate all tree structures, it is sufficient to provide for `info` at most as many values as there are nodes. Also, it is sufficient to provide only non-negative values for `size`: a valid tree cannot have a negative number of nodes. The user can use Korat itself to "debug" the finitization: if Korat generates no valid structure, it may be due to the finitization (or the predicate). Also, if Korat generates too many valid structures, it may be possible to reduce the number by changing the finitization.

3.2.3 State Space

Korat constructs a state space of predicate structures based on a finitization. Korat first allocates the objects that appear in the finitization and then generates a vector of possible values for each field of those objects. Each *state* assigns a particular value to each of the fields, and the whole *state space* consists of the Cartesian product of possible values for each field.

For example, consider the finitization that `finSearchTree(3)` (from Figure 3-6) produces. Korat first allocates one `SearchTree` object that forms the `SearchTree` class domain and three `Node` objects that form the `Node` class domain. To systematically explore the state space, Korat orders the objects in these domains and during search uses indexes into these domains. Figure 3-7 shows the classes that represent the data for the state space and search.

Korat next assigns a field domain to each field. In this example, there are 11 fields: the single `SearchTree` object has two fields (`root` and `size`) and the three `Node` objects have three fields each (`left`, `right`, and `info`). Each field domain is a sequence of class domain indexes, such that all values that belong to the same class domain occur consecutively. For example, the field domain for `root` has four elements, `null` and three `Node` objects, where `null` (as well as each primitive value) forms a class domain by itself. Therefore, the field domain for `root` is `[null,<nd,0>,<nd,1>,<nd,2>]`, where `nd` is the class domain for `Node` objects.

Figure 3-8 shows the state space for this example. Each state is a mapping from the object fields to the field domain indexes, and the whole state space consists of all possible mappings. Since the domains for `root`, `left`, and `right` have four elements,

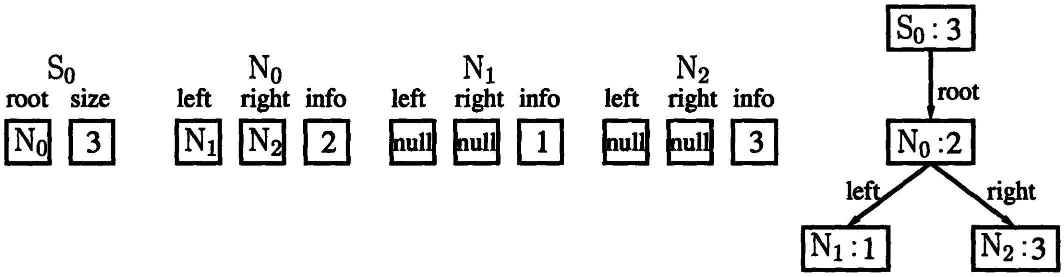

Figure 3-9: Candidate that is a valid SearchTree.

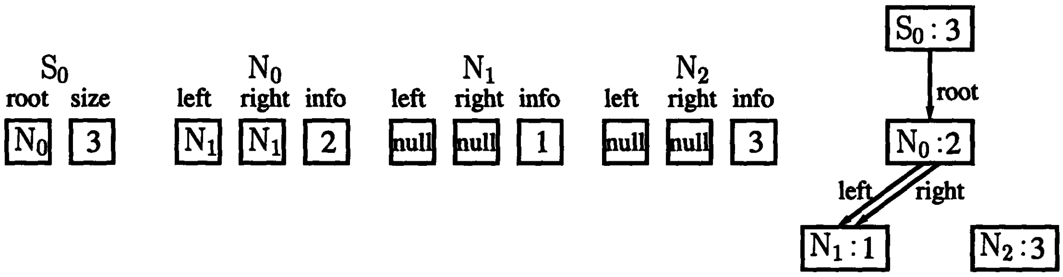

Figure 3-10: Candidate that is not a valid SearchTree.

the domain for size has four elements, and the domain for info has three elements, the state space has $4 \cdot 4 \cdot (4 \cdot 4 \cdot 3)^3 = 1769472 > 2^{20}$ states in this example. For scope= n, the state space has $(n+1)^{2(n+1)} \cdot n^n$ states.

Each state encodes a candidate structure. While a state maps fields to field domain indexes (which are integers), a candidate is a structure that consists of the Java objects from the finitization. Each field in the candidate is set according to the field domain indexes in the state. Korat builds states for the systematic search of the state space, and it builds candidates for actual inputs to the predicate. Because of the bijection between states and candidates, we use terms *state* and *candidate* interchangeably. We define two states to be isomorphic iff the corresponding candidates are isomorphic (Definition 4).

Figure 3-9 shows an example candidate that is a valid binary search tree with three nodes. Not all candidates represent valid binary search trees. Figure 3-10 shows an example candidate that is not a tree; repOk returns false for this candidate. Assume that the field domains are ordered as follows: for left and right the same as for root (first null then Node objects), for size [0,1,2,3], and for info [1,2,3]. Then, the candidate in Figure 3-9 (Figure 3-10) corresponds to the state [1,3,2,3,1,0,0,0,0,0,2] ([1,3,2,2,1,0,0,0,0,0,2]).

Connected Structures

We next determine an approximate number of different connected substructures in our example state space. This number gives a better measure than the size of the state space (due to Korat's encoding) for comparing Korat's search with other plausible searches. We call a structure *connected* if all objects are reachable from the root. A random search, for instance, could generate structures as graphs reachable from root, not by randomly choosing a state from the state space. Each structure $\langle O, r, E\rangle$ contains a connected substructure $\langle O_c, r, E_c\rangle$, where $O_c = \text{reachable}(\langle O, r, E\rangle)$ is the set of all reachable objects, and $E_c = \{\langle o, f, o'\rangle \in E | o \in O_c\}$. For example, every state that maps the `root` field of the `SearchTree` object to `null` contains the connected substructure that has only the `SearchTree` object and no `Node` objects.

We have determined that our example state space has $(n+1)^{2(n+1)} \cdot n^n$ states for the scope n. Different states from the state space represent different structures, but may have the same connected substructure (i.e., the same values for all fields of all objects reachable from the root), and only differ in the values of some field of an object unreachable from the root. For example, two structures that have the value of `root` field `null` (and the same value for the `size` field) have the same connected substructure, although they can differ in values for the fields of `Node` objects.

In our example, we compute the number of connected structures for the scope n, denoted C_n, as the sum of the numbers of connected structures that have exactly i objects, denoted $C_{i,n}$, where i ranges from 0 to n, i.e., $C_n = \sum_{i=0}^{n} C_{i,n}$. The state space includes both valid and invalid structures; we want to compute the number of connected structures (i.e., graphs) and not only the number of valid structures (i.e., trees). Let G_i be the number of nonisomorphic underlying graphs for the connected structures with i objects such that $C_{i,n} = G_i \cdot i! \cdot \binom{n}{i} \cdot (n+1) \cdot n^i$: each graph can have any permutation of i objects, chosen from a set of n objects; `size` can have one of $n+1$ values, and each `info` in the i nodes can have one of n values.

To determine an approximation for C_n, we determine an approximation for G_i, the number of rooted, connected, directed, node-unlabeled graphs with i nodes and two edge labels (`left` and `right`). We can represent each such graph with a rooted spanning tree and a set of back edges; we compute G_i by computing the number of trees and the number of back edges that can be added to each tree. The number of rooted trees with exactly i nodes, denoted T_i, obeys the recurrence $T_i = \sum_{j=0}^{i-1} T_j T_{i-1-j}$ with $T_0 = 1$; to get a tree with i nodes, we choose one node for the root and split the other $i-1$ nodes in the left and right subtrees. This is a standard recurrence [101] that gives rise to Catalan numbers: $T_i = \frac{1}{i+1}\binom{2i}{i}$.

Each of T_i trees has i nodes and $i-1$ edges, effectively setting the values for $i-1$ out of $2i$ fields. Each graph can set the values for the remaining $i+1$ fields to one of $i+1$ values, either to be `null` or to point to one of the i nodes. We have that $G_i \leq (i+1)^{i+1} T_i$: setting the fields to all possible values for all trees creates the same graph from different (spanning) trees. We also have that $G_i \geq 2 \cdot 3 \cdot \ldots \cdot (i+1) \cdot (i+1) T_i$; we can show that setting the fields according to the following procedure does not create the same graph from different trees: (1) assign to each node a number between 1 and i based on the node's order in a traversal such as depth-first traversal; (2) set a field for node j to be either `null` or to point to node $j' \leq j$. With this procedure, there is at least one field that has two possible values, one that has three etc.

We have not determined the exact formula for G_i. Using Korat, we have found the exact values for G_i for several i, starting from 0: 1; 4; 45; 816; 20,225; 632,700; 23,836,540... This sequence does not appear in the Sloane's On-Line Encyclopedia of Integer Sequences [98], so it may have not arisen in combinatorics research previously. We conjecture that the number of graphs is:

$$G_i = (i+1)^2 \sum_{\substack{\langle a_1,\ldots,a_{i-1}\rangle \\ i+1 \geq a_{i-1} \geq \ldots \geq a_1 \land \forall 1 \leq j \leq i-1.\, a_j \geq j+1}} \prod_{j=1}^{i-1} a_j.$$

In summary, we obtain the following lower bound for the number of connected structures: $G_i \geq (i+1) \cdot (i+1)! \cdot \frac{1}{i+1}\binom{2i}{i} = \frac{(i+1)(2i)!}{i!}$; $C_{i,n} \geq (i+1)(2i)!\binom{n}{i}(n+1)n^i$, and $C_n = \sum_{i=0}^{n} C_{i,n}$. We use this lower bound to under-approximate the size of the structure space for `SearchTree` in our evaluation of Korat's search (Section 8.2). We derive similar lower bounds for other benchmarks.

3.2.4 Search

Figure 3-11 shows a detailed pseudo-code of the Korat's search algorithm. Based on the finitization, Korat first creates the state space and gets the root object of all candidates and structures. It then initializes `result`, the set of (valid) structures found so far in the search, to the empty set. During the search, Korat maintains a `stack` of fields that the executions of the predicate access; this stack is initially empty.

The search starts with the state set to all zeros. For each state, Korat first creates the corresponding candidate structure by setting the fields according to the indexing of the state into the state space. Korat then executes the predicate to check the validity of the candidate.

During the execution, Korat monitors the fields that the predicate accesses and updates the `stack` in the following way: whenever the predicate accesses a field that is not already on the stack, Korat pushes the field on the stack. This ensures that the fields in the stack are ordered by the first time the predicate accesses them, i.e., the predicate has accessed (for the first time) the fields near the top of the stack after it had accessed (for the first time) the fields near the bottom of the stack. The predicate may access the same field several times, but the order is with respect to the first access.

As an illustration, consider that the stack is empty and Korat invokes `repOk` on the candidate shown in Figure 3-10.[2] In this case, `repOk` accesses only the fields [`root`,N_0.`left`,N_0.`right`] (in that order) before detecting a cycle and returning `false`. Thus, after this invocation, the stack would consist of only those three fields.

[2] This candidate is just an illustrative example for the search; the Korat search would actually skip over this candidate without invoking `repOk` on it.

```
Set<Map<ObjField, int>> koratSearch(Predicate pred, Finitization fin) {
  Map<ObjField, ClassDomainIndex[]> space = fin.getSpace();
  Object root = fin.getRootObject();
  Set<Map<ObjField, int>> result = new Set();
  Stack<ObjField> stack = new Stack();
  Map<ObjField, int> state = new Map();
  foreach (ObjField f in fin.getObjFields()) state[f] = 0;
  do {
    // create a candidate
    foreach (ObjField f in fin.getObjFields()) {
      ClassDomainIndex cdi = space[f][state[f]];
      f.set(cdi.domain.objects[cdi.index]);
    }
    // execute "pred(root)" and update "stack"
    boolean value = observeExecution(pred, root, stack);
    // if state is valid, add it to the result
    if (value) result.add(state);
    // if *not* optimizing, add other fields to the stack
    if (!PRUNING || result) {
      // add all reachable fields not already in stack
      foreach (ObjField f in reachableObjFields(root))
        if (!stack.contains(f)) stack.push(f);
    }
    // backtrack
    while (!stack.isEmpty()) {
      ObjField f = stack.top(); // field on the top of stack
      if (ISOMORPHISM_BREAKING) {
        int m = -1; // maximum index for the class domain of 'f'
        ClassDomain d = space[f][state[f]].domain;
        // a straightforward way to compute 'm'
        foreach (ObjField fp in stack.withoutTop())
          if (space[fp][state[fp]].domain == d)
            m = max(m, space[fp][state[fp]].index);
        // if an isomorphic candidate would be next...
        if (space[f][state[f]].index > m)
          // ...skip to the end of the class domain
          while ((state[f] < space[f].length - 1) &&
                 (space[f][state[f] + 1].domain == d))
            state[f]++;
      }
      // check if the field index is at the end of the field domain
      if (state[f] < space[f].length - 1) {
        state[f]++; break; // increment index and stop backtracking
      } else {
        state[f] = 0; stack.pop(); // reset index and keep backtracking
      }
    }
  } while (!stack.isEmpty()); // end do
  return result;
}
```

Figure 3-11: Pseudo-code of the Korat's search algorithm.

If the predicate returns `true`, Korat adds the current state to the set of valid structures. It also ensures that all reachable fields are on the stack, so that successive iterations generate all (nonisomorphic) states that have the same values for the accessed fields as the current state.

Korat then uses backtracking to generate the next state based on the accessed fields. We explain the isomorphism optimization in the next subsection; we focus here on the code that checks for the end of the field domain. Korat first checks whether it can increment the field domain index for the field on the top of the stack without exceeding the field domain size. If it can, Korat increments this field domain index and finishes the current iteration. If it cannot, Korat resets this field domain index to zero and moves to the previous field on the stack, unless the stack is empty. (The isomorphism optimization may additionally skip more values from a field domain.)

Continuing with our example, the next candidate gets the next value for N_0`.right`, which is N_2 by the above order, and the values of the other fields do not change. This prunes from the search all $4^5 \cdot 3^3 = 27648$ states of the form `[1,_,2,2,_,_,_,_,_,_,_]` that have the (partial) valuation: `root=`N_0, N_0`.left=`N_1, N_0`.right=`N_1. Intuitively, the pruning based on accessed fields does not rule out any valid structure because `repOk` did not read the other fields, and it could have returned `false` irrespective of the values of those fields.

Note that the execution of `repOk` on the next candidate starts with the content of the stack that the execution on the previous candidate finishes with. More specifically, the stack is not initialized to empty between iterations (but only once before the main `do` loop). If `repOk` accesses fields in a deterministic order, it would be the same if the stack were initialized to empty; since the backtracking does not change the values of the first several fields on the stack, the execution of `repOk` on the next candidate would build the same stack for those fields.

Continuing further with our example, the next candidate is the valid binary search tree shown in Figure 3-9. When Korat executes `repOk` on this candidate, `repOk` returns `true` and the stack contains all 11 fields. (In general, `repOk` may not access all reachable fields before returning `true`.) The search then backtracks to the next state.

Korat repeats the procedure with the next state: generate the corresponding candidate, invoke the predicate on it, monitor the execution to update the stack, and backtrack on the fields in the stack. The search finishes when the stack is empty after backtracking, indicating that the whole state space has been explored. The result of the search is the set `result`.

3.2.5 Nonisomorphism

Recall from Section 3.1.2 that isomorphism is defined with respect to node identities, and since structures, and thus candidates and valid structures, are rooted and deterministically edge-labeled graphs, it is easy to check isomorphism. However, Korat does not explicitly check isomorphism to filter out isomorphic structures. Instead, Korat avoids generating isomorphic structures by not even considering isomorphic candidates. The code that implements this is shown in Figure 3-11 in the if (ISOMORPHISM_BREAKING) part. This code depends on structures being the inputs to the predicate; if the inputs were general graphs, Korat could generate some isomorphic inputs.

Recall that each state maps object fields to field domain indexes. Field domain indexes have a natural order. Additionally, each stack imposes a partial order on the fields: a field f_1 is "smaller" than a field f_2 iff either (1) f_1 is on the stack and f_2 is not on the stack or (2) both f_1 and f_2 are on the stack and f_1 is farther from the top of the stack than f_2; two fields that are both not on the stack are incomparable. For example, the stack [root,N_0.left,N_0.right] makes root "smaller" than any other field, N_0.left "smaller" than any field but root, N_0.right "smaller" than all fields that are not on the stack, and all those fields that are not on the stack incomparable.

The two orders, the order on field domain indexes and the partial order on fields, together induce a partial *lexicographic order* on the states: this lexicographic order first orders the fields in a sequence based on the order on fields and then compares the field domain indexes within the sequence as usual for the lexicographic order. Korat generates states in the increasing values of this lexicographical order. For example, for the stack [root,N_0.left,N_0.right], Korat generates the state(s) that have the (partial) valuation root=N_0, N_0.left=N_1, N_0.right=N_1 before the state(s) that have the (partial) valuation root=N_0, N_0.left=N_1, N_0.right=N_2.

With the isomorphism-breaking optimization, Korat avoids generating isomorphic states. More precisely, Korat generates only the lexicographically smallest state from each isomorphism partition from which Korat generates at least one state. (Due to pruning, Korat may not generate any state from some isomorphism partitions; those partitions, however, do not have any valid state.) Conceptually, Korat avoids isomorphic states by incrementing some field domain indexes by more than one. (The usual backtracking increments field domain indexes at most by one.)

For the field f on the top of the stack, Korat finds m, the maximum class domain index of all fields fp that are "smaller" than f (farther from the top of the stack) and have the same class domain as f. If there is no such fp, m has value -1. (The actual implementation caches the values of m and reuses them in next iterations, because a part of the state remains the same between backtracking iterations.) For example, for the state from Figure 3-10, and with the stack as discussed above, m=1 for f=N_0.right.

While backtracking on f, Korat checks if the field domain index for f is greater than m. If it is not, Korat does nothing special for f but only increments its field domain index as usual in backtracking. This allows Korat to explore all possible aliasing of the field f with "smaller" fields. However, if the field domain index is greater than m, Korat increments this field domain index to the end of the current class domain for f. This allows Korat to prune all but one states where f would point to an object not pointed by any of the "smaller" fields. Intuitively, this pruning does not rule out any valid structures, but only avoid isomorphic structures.

For example, Korat for finSearchTree(3) generates only the 14 trees shown in Figure 2-2. Consider the five trees of size three. Each of them is a representative from an isomorphism partition that has six distinct trees, one for each of 3! permutations of nodes.

In general, if each valid structure for some repOk predicate has all n objects from a class domain, Korat with the isomorphism optimization reduces the number of structures generated by a factor of $n!$. This factor is smaller when valid structures can have repeated objects and/or do not need to have all objects from some domain. For example, suppose that Korat generates an object with three fields whose field domains consist of only one class domain $\{o_1, o_2, o_3, \ldots, o_n\}$ and repOk always returns true. Out of n^3 potential candidates, Korat generates only the following five, one from each partition of nonisomorphic inputs: o_1, o_1, o_1; o_1, o_1, o_2; o_1, o_2, o_1; o_1, o_2, o_2; o_1, o_2, o_3.

To illustrate the importance of eliminating isomorphic candidates during generation, consider the alternative technique that first generates all valid structures (irrespective of isomorphism) and then filters out isomorphic structures, keeping only one structure from each isomorphism partition. Suppose that a finitization has only one class domain with n objects and that there are $V(n)$ valid nonisomorphic structures in each of which the n objects appear exactly once. The alternative technique would first generate $n!$ more candidates and valid structures than Korat, i.e., $n! \cdot V(n)$ valid structures. If it then eliminates isomorphic structure by comparing each structure with all others not already eliminated, it would additionally require $\Sigma_{i=1}^{V(n)}(n!(V(n) - i + 1) - 1) = O(n! \cdot V(n))$ comparison steps between the structures.

3.2.6 Implementation

We next present some details of our Korat implementation. The implementation has three major parts: (1) the search itself, (2) generation of finitization skeletons, and (3) instrumentation for monitoring field accesses.

We implemented in Java the search for valid nonisomorphic structures, closely following the algorithm from Figure 3-11. The main difference is that the implementation does not represent object fields using the ObjField class (a pair of an object and a field); instead, it assigns a unique integer identifier to each field of the objects in the finitization. This is a performance optimization that allows representing state as an array that maps field identifiers into field domain indexes.

We modified the source code of the Sun's javac compiler to provide automatic generation of finitization skeletons. Given a set of Java source files, our implementation uses the compiler to parse them and build symbol tables; it then processes the type information to generate skeletons as illustrated in Section 3.2.2. The modified javac generates Java source of the skeletons.

Instrumentation

We also modified the javac compiler to provide instrumentation for monitoring field accesses. Our implementation performs a source-to-source translation that instruments all classes whose objects appear in finitizations. For each class, the instrumentation adds a special constructor and a field for the observer (explained below). For each field of these classes, the instrumentation adds an identifier field and special get and set methods. Finally, in the code for repOk and all methods that repOk transi-

```
class SearchTree {
    korat.Observer korat_observer; // observer for this object
    // special constructor to initialize the observer and field ids
    SearchTree(korat.Observer observer, korat.Ids id) {
        korat_observer = observer;
        korat_id_root = id.getNextId();
        korat_id_size = id.getNextId();
    }
    Node root; // root node
    int korat_id_root; // identifier for the root field
    // special get method for the root field
    Node korat_get_root() {
        korat_observer.notify_get(korat_id_root);
        return root;
    }
    // special set method for the root field
    void korat_set_root(Node value) {
        korat_observer.notify_set(korat_id_root);
        root = value;
    }
    int size;  // number of nodes in the tree
    int korat_id_size; // identifier for the size field
    // special get method for the size field
    int korat_get_size() {
        korat_observer.notify_get(korat_id_size);
        return size;
    }
    // special set method for the size field
    void korat_set_size(int value) {
        korat_observer.notify_set(korat_id_size);
        size = value;
    }

    // in repOk (and isTree, numNodes, isOrdered and other transitively
    // invoked methods) replace field accesses with get and set methods
    boolean repOk() {
        // checks that empty tree has size zero
        if (_korat_get_root() == null) return _korat_get_size() == 0;
        // checks that the object graph is a tree
        if (!isTree()) return false;
        // checks that size is consistent
        if (numNodes(_korat_get_root()) != _korat_get_size())
            return false;
        // checks that data is ordered
        if (!isOrdered(_korat_get_root())) return false;
        return true;
    }
}
```

Figure 3-12: Instrumentation for the SearchTree example.

tively invokes, the instrumentation replaces each field read/write with an invocation of the corresponding get/set method. Our implementation similarly instruments arrays, essentially treating each array element as a field.

Recall that Korat monitors executions of repOk (and transitively invoked methods) to build the stack of accessed fields. The instrumentation achieves this using an approach similar to the *observer* pattern [35]. Figure 3-12 illustrates the instrumentation on the SearchTree class from Figure 2-1. (The instrumentation similarly modifies the Node class.) The special constructor initializes the korat_observer field and the identifiers for all instance fields, in this case root and size. The finitization uses the special constructors to initialize all objects with an observer and to initialize each identifier field to a unique integer. Besides the identifier, each of the fields has korat_get and korat_set methods. During executions, these methods first notify the observer that a field, with a given field identifier, is being accessed and then perform the actual access, i.e., return the field's value for get or assign the value to the field for set. Finally, the instrumentation of the code for repOk replaces field reads with invocations of these methods.

In the example repOk, there are no field writes. This is a common case: most practical predicates are pure, i.e., they do not modify the candidate structure. (The predicates can still create and modify other objects, e.g., isTree modifies the set visited.) The Korat search algorithm does not require repOk to be pure. However, we optimized our implementation for this common case: if the predicate does not modify the candidate, then generating the next candidate from the next state requires setting only the values of the field(s) that were backtracked on, not all fields.

Chapter 4

Correctness

This chapter presents a proof of correctness for the Korat search algorithm presented in the previous chapter. Section 4.1 discusses some of the challenges in proving the correctness of the search. Section 4.2 introduces the notation used in the proof. The search takes as input a predicate and a finitization. Section 3.2.2 has already described finitizations. Section 4.3 states the conditions about the predicates that the search operates on. Section 4.4 then states the correctness requirements for the search: it should be sound (generate only valid structures), complete (generate all nonisomorphic valid structures), and isomorph-free (generate only nonisomorphic structures). Section 4.5 finally proves that the Korat search indeed satisfies these requirements for all predicates that satisfy the conditions. This proof provides a strong guarantee that should increase the confidence of Korat users in the correctness of a test-input generation tool based on Korat.

4.1 Challenges

There are several properties of the Korat search that complicate proving its correctness. Similar properties would be inherent in any search that uses pruning to improve on the naive exploration of the entire state space (Section 3.1.3). The additional features of Korat are that it applies isomorphism-breaking optimization and handles nondeterministic predicates. We next describe the challenging properties of the Korat search.

State encoding: For efficiency purposes, Korat encodes structures using states that map to integers indexing into the finitization. This creates a level of indirection: the search operates on states, whereas the correctness requirements involve validity and isomorphism that are defined for structures (Section 3.1.2).

Field ordering: Korat searches the state space based on a *dynamic* order in which the predicate executions access the fields. Our Korat implementation assigns field identifiers to fields statically (i.e., independent of the predicate executions), but the predicate can access these fields in any order.

Isomorphism optimization: Korat generates only nonisomorphic structures, but does so without explicitly checking isomorphism of pairs of structures. Instead, the isomorphism optimization prunes some structures from the search, and the proof thus needs to show that the pruning does not eliminate any nonisomorphic valid structure.

Nondeterministic predicates: Korat generates all structures for which the predicate always returns true and no structure for which the predicate always returns false; Korat may or may not generate the other structures. When predicates are deterministic, then they always produce the same result for the same structure and intuitively have the "same prefix" of the execution for the "same prefix" of the structure. For deterministic predicates, the search algorithm would not need to preserve the `stack` across iterations, as discussed in Section 3.2.4.

4.2 Notation

We use the meta-variable π to range over predicates. We use γ and γ' to range over structures that are inputs to the predicate. We write $\pi(\gamma)$ for the set of results that executions of π can produce for structure γ. Recall that γ is valid iff $\pi(\gamma)=\{\texttt{true}\}$, and γ is invalid iff $\pi(\gamma)=\{\texttt{false}\}$ (Definition 3).

We use σ to denote the value of the `stack` of fields that the predicate executions access (Figure 3-11). We write len(σ) for the length of the stack σ and $\sigma(i)$ for the field at offset i, where 0 is the offset at the bottom of σ, and len$(\sigma) - 1$ is the offset at the top of σ.

We use Σ and Σ' to range over states. Each state maps fields to field domain indexes. We use f to range over fields and ϕ to range over sets of fields. We write $\Sigma(f)$ for the field domain index of the field f in state Σ.

We define a partial order between states based on the lexicographic order of field domain indexes of those fields that are on the stack. Let Σ and Σ' be two states and σ be a stack. We say that Σ is σ-smaller than Σ', in notation $\Sigma <_\sigma \Sigma'$, iff $\exists i < \text{len}(\sigma).\ \Sigma'(\sigma(i)) < \Sigma(\sigma(i)) \wedge \forall i' < i.\ \Sigma(\sigma(i')) = \Sigma'(\sigma(i'))$.

4.3 Conditions

We next precisely list the conditions that a predicate needs to satisfy to be an input to the Korat search. If a predicate does not satisfy some of the conditions, then the search may not generate all nonisomorphic valid structures for the predicate.

Let Π be the set of predicates such that all executions of each $\pi \in \Pi$ satisfy the following conditions:

C1. Each execution terminates, returning `true` or `false`.

C2. Each field that the execution accesses is either (1) a part of the input structure or (2) a field of some object that the predicate locally allocates. In other words, the execution does not access global data through static fields.

C3. No execution depends on the actual allocation address of objects. (This holds for Java predicates if no execution invokes `System.identityHashCode` method.)

The user of Korat needs to write the predicate so that Condition C1 holds. It implies that the predicate should terminate for all structures, be they valid or invalid. For example, `repOk` for `SearchTree` checks that the input object graph is actually a tree and terminates (returning `false`) if there is a cycle in the graph. A static or dynamic analysis can easily check the other two conditions, C2 and C3; we have found all practical predicates to satisfy these two conditions, and Korat does not currently check them.

The Korat search operates on Java predicates, so to prove that the search is correct, we need in principle to reason about the executions of Java predicates. However, Java is a complex programming language, and researchers have formalized only parts of its semantics [54]. We thus consider a simple model where Java predicates are state machines [66]. Within this model, we sketch generic proofs for the following two properties of executions:

1. The execution of a predicate does not depend on the fields that the predicate does not access.

2. Given isomorphic structures, a predicate produces the same result.

In general, these two properties are assumptions on the executions, and we should prove them for the language in which we write the predicates. We present our results in the context of Java, but they generalize to other languages whose executions satisfy these two properties.

We first show that the execution of a predicate does not depend on the fields that the predicate does not access.

Lemma 1. *Consider two candidates γ and γ' that have identical values for all fields from some set ϕ. If an execution of π with structure γ accesses only fields from ϕ before returning a result, then there exists an execution of π with structure γ' that accesses the same fields and returns the same result.*

Proof. Proceed by induction on the length of the execution of the predicate for γ. As the witness execution for γ', choose the execution that makes the same steps as the execution for γ, i.e., the execution for γ' makes the same nondeterministic choices as the execution for γ. At each corresponding step of these two executions, the states have identical values for all fields from ϕ, because no step accesses a field that is not in ϕ. □

The following is a simple corollary.

Corollary 1. *If two candidates have identical values for more fields than in ϕ, these two candidates have the same set of executions.*

We next show that any predicate from Π returns the same result for isomorphic input structures. This is a property of the programming language used for predicates; we again sketch a generic proof that we need to instantiate with respect to semantics of each specific language.

Lemma 2. *For all isomorphic γ and γ', $\pi(\gamma) = \pi(\gamma')$.*

Proof. We need to show that for each execution of π with input γ, there exists an execution of π with input γ' such that the two executions generate the same result. Proof proceeds by induction on the length of execution for γ. As the witness execution for γ', choose the execution that makes the same steps as the execution for γ. By Condition C2, no execution of π accesses a field from the finitization that is not reachable from the root object (the input to the predicate). Thus, at each corresponding step of these two executions, the states have isomorphic values for all fields (1) reachable from the root object or (2) belonging to the objects locally allocated. Further, by Condition C3, no step depends on the object identity, so the states are isomorphic for these fields and in the final state, the executions return the same result. □

It follows that isomorphic inputs have the same (in)validity. This is important because it allows Korat to consider only one candidate from each isomorphism partition: if this candidate is (in)valid, then all other candidates from the same partition are also (in)valid.

4.4 Statement

Given a predicate and a finitization, the Korat search generates a set of structures. More precisely, Korat generates all nonisomorphic valid structures, within the given finitization. This set of structures depends on the results that the predicate returns for the candidate structures. If the predicate is nondeterministic in result, Korat can generate different sets of structures for the same predicate and finitization. We prove that all these sets satisfy the soundness, completeness, and isomorph-freeness properties required for a solver for imperative predicates (Section 3.1.2).

Theorem 1 (Korat Correctness). *For every predicate $\pi \in \Pi$ and for every set of structures that the Korat search generates for π:*

- ***Soundness:** Korat is sound: the set contains no invalid structure.*

- ***Completeness:** Korat is complete: the set contains at least one structure from each isomorphism partition of valid structures.*

- ***Isomorph-Freeness:** Korat is isomorph-free: the set contains at most one structure from each isomorphism partition.*

4.5 Proof

The proof consists of three lemmas that show the soundness, completeness, and isomorph-freeness. These three lemmas directly imply the main theorem and thus show that Korat is a correct solver for imperative predicates (from the set Π).

A search is sound if it generates no invalid structure.

Lemma 3 (Soundness). *Korat does not generate an invalid structure for any predicate (even if not from* Π*).*

Proof. By contradiction; suppose that Korat generates an invalid structure γ for some predicate π. It means that all executions of π for input γ return `false`. However, the algorithm in Figure 3-11 generates γ (i.e., adds γ to the set `result`) only if an execution of π returns `true`. Contradiction! □

A search is complete if it generates at least one valid structure from each isomorphism partition. To prove completeness of Korat, we first consider Korat*, the Korat search with the argument `PRUNING` set to `false`.

Lemma 4. *If Korat* is complete for some predicate from* Π*, then Korat is also complete for that predicate.*

Proof. Assume that Korat* is complete, i.e., it always generates at least one valid structure from each isomorphism partition. Korat generates a structure if it is a candidate input for an execution that returns `true`. For a valid structure, all executions return `true`, so Korat generates a valid structure if it is a candidate for any execution. Since Korat* generates at least one valid structure from each isomorphism partition, it also considers as a candidate at least one valid structure from each isomorphism partition. Due to pruning, Korat considers less candidates than Korat*. We will show that Korat still considers as a candidate at least one valid structure from each isomorphism partition. Thus, Korat generates at least one valid structure from each isomorphism partition, i.e., Korat is complete.

By contradiction, suppose that Korat does not consider as a candidate any valid structure from some isomorphism partition. Since Korat* considers such a structure, it must be that Korat prunes this valid candidate structure γ. This pruning occurs after the predicate executes some candidate structure γ' and returns `false`. Let the `stack` after the execution of γ' be σ. Before returning `false`, this execution has accessed only (some of the) fields from σ. Further, it is easy to show that Korat* prunes only candidate structures that have the same valuation for all fields in σ as γ', i.e., $\forall f \in \sigma . \gamma(f) = \gamma'(f)$. By Corollary 1, since there exists an execution for γ' that returns `false`, there also exists an execution for γ that returns `false`. This contradicts the assumption that γ is valid, i.e., all executions for structure γ return `true`. □

Lemma 5. *Korat* is complete for all predicates from* Π.

Proof. (Sketch) We need to show that Korat* generates at least one valid structure from each isomorphism partition. It is sufficient to show that Korat* considers as a candidate at least one valid structure from each isomorphism partition.

The proof proceeds by induction on the number of considered candidates, i.e., the number of iterations of the main loop of Korat*. Each iteration consists of a predicate execution with potential adding of fields to the stack and backtracking with isomorphism breaking. Let σ and Σ be the values of `stack` and `state`, respectively, after backtracking. Let Γ be the set of candidates considered up to that iteration.

Recall the ordering between states $\Sigma <_\sigma \Sigma'$. Let s be: (1) the set of all states σ-greater than Σ ($\{\Sigma' | \Sigma <_\sigma \Sigma'\}$) if σ is not empty and (2) the set of all states if σ is empty. An easy induction can show that Γ contains at least one representative from each isomorphism partition that has a representative in s. Since the search terminates when the stack becomes empty, it follows that the search considers at least one candidate structure from each isomorphism partition of candidates. □

Lemma 6 (Completeness). *Korat is complete for all predicates from* Π.

Proof. Follows from Lemma 4 and Lemma 5. □

We finally prove isomorph-freeness. Recall that we define structures as rooted, deterministically edge-labeled graphs (Definition 2) and isomorphism with respect to object identity (Definition 4). If the inputs to the predicate were arbitrary graphs, Korat would not eliminate all isomorphic inputs.

Lemma 7 (Isomorph-Freeness). *Korat is isomorph-free for all predicates from* Π.

Proof. Let Γ be the set of candidate structures for which Korat executes the predicate. Since the result of Korat is a subset of Γ, it suffices to show that the search executes the predicate for at most one structure from each isomorphism partition. By contradiction; suppose that there are two isomorphic distinct candidate structures, γ and γ'. Let p be a permutation between these candidates. Consider the stacks, σ and σ', after the backtracking for candidates γ and γ', respectively. Let σ_0 be the common prefix for these two stacks. There are two cases:

- For some field in σ_0, the candidates have different values. Let i be the index such that
$$\gamma(\sigma_0(i)) \neq \gamma'(\sigma_0(i)). \tag{4.1}$$
and
$$\forall i' < i . \gamma(\sigma_0(i)) = \gamma'(\sigma_0(i)). \tag{4.2}$$
Since γ and γ' are isomorphic,
$$\forall i \in \text{len}(\sigma) . p(\gamma(\sigma_0(i))) = \gamma'(\sigma_0(i)). \tag{4.3}$$
From 4.2 and 4.3, we have that p is identity for all $i' < i$:
$$\forall i' < i . p(\gamma(\sigma_0(i'))) = \gamma(\sigma_0(i')). \tag{4.4}$$
We next consider two cases based on the value for the i-th field already appearing before in the stack:
$$\exists i' < i . \gamma(\sigma_0(i)) = \gamma(\sigma_0(i')). \tag{4.5}$$
  - If 4.5 holds, let i' be the previous index. We have $p(\gamma(\sigma_0(i))) = p(\gamma(\sigma_0(i')))$. From 4.4, we have that $p(\gamma(\sigma_0(i))) = \gamma(\sigma_0(i'))$. Further, again from 4.5, we have $p(\gamma(\sigma_0(i))) = \gamma(\sigma_0(i))$ and then from 4.3 $\gamma'(\sigma_0(i)) = \gamma(\sigma_0(i))$, which contradicts 4.1.

  - If 4.5 does not hold, then $\forall i' < i.\gamma(\sigma_0(i)) \neq \gamma(\sigma_0(i'))$. Let $\text{mIndex}(\gamma, \sigma_0, i)$ be the unique value m that the search computes in `m`. For field $\sigma_0(i)$, the search generates values up to m. Since $\gamma(\sigma_0(i))$ is different than $\gamma(\sigma_0(i'))$ for all $i' < i$, it means that $\gamma(\sigma_0(i)) = m$, and this is the only such candidate. Thus, $\gamma'(\sigma_0(i)) = m$ also, and again $\gamma(\sigma_0(i)) = \gamma'(\sigma_0(i))$; contradiction!

- For all fields in σ_0, the candidates have the same value. There are two cases: either σ_0 is a true prefix of σ and σ' or σ_0 is the same as σ or σ'.

  - If σ_0 is the same as one of the stacks, assume w.l.o.g. that $\sigma_0 = \sigma$. Consider the smallest index i such that $\gamma(\sigma'(i)) \neq \gamma'(\sigma'(i))$; there must be such a field since $\gamma \neq \gamma'$. (More precisely, this value is not `0` in γ'; it is `0` in γ, because it is not in the stack for γ.) Since the search generates γ', it means that the for all indexes i of σ', the values of the fields $\gamma'(\sigma'(i)) \leq \text{mIndex}(\gamma', \sigma', i)$. Further, for all $i' \leq i$, fields $\gamma'(i')$ are reachable in both γ and γ'. However, one of those fields, $\sigma'(i)$, has different values in γ and γ': $\gamma(\sigma'(i)) < \gamma'(\sigma'(i)) \leq \text{mIndex}(\gamma', \sigma', i)$. Thus, γ and γ' are not isomorphic; contradiction!
  - If σ_0 is not the same as any of the stacks, then the two stacks have a different field right after σ_0. Assume w.l.o.g. that the search first generates the stack σ. The only way to change the value of the field after σ_0, say f, is during backtracking if the search tries all possibilities for f and then backtracks on the previous field, the last in σ_0. This backtracking, however, changes the value of this last field in σ_0. Therefore, σ_0 cannot be the common prefix for σ and σ', if the candidates have the same values for all fields in σ_0. Contradiction!

This cover all the cases, and in each of them gives a contradiction if we suppose that the search considers two distinct, isomorphic candidate inputs. Hence, Korat is isomorph-free. □

Chapter 5

Direct Generation

This chapter discusses direct generation, an approach for obtaining structures that satisfy properties of interest by writing programs that directly generate such structures. We call such programs generators. Section 5.1 presents an example generator. Section 5.2 then compares generators and imperative predicates: whereas generators directly generate structures of interest, imperative predicates only check the properties of interest, and to obtain the actual structures from imperative predicates, we need to solve them, for example with Korat. We argue that for complex properties, imperative predicates are easier to write than generators that generate all nonisomorphic valid structures within a given bound. Moreover, imperative predicates are useful for more testing tasks than generators. However, generators can generate structures much faster than solvers for imperative predicates. Section 5.3 thus presents dedicated generators, an extension to Korat inspired by generators. Dedicated generators enable faster solving and also make it easier to write imperative predicates.

5.1 Example

We use the binary search tree example from Section 2.1 to illustrate generators. Recall the Java code from Figure 2-1. Each `SearchTree` object has a field `size` and a pointer `root`. Each tree consists of several linked nodes. Each `Node` object has an element `info` and pointers to children `left` and `right`. Recall further that a tree is in a scope s if it has at most s nodes whose elements range between 1 and s. We next proceed to develop a generator for all nonisomorphic binary search trees in scope s.

For the beginning, instead of generating all binary search trees, suppose that we just want to randomly generate some object graphs that are trees (i.e., have no sharing) regardless of the values in the nodes. This is seemingly easy: we just make each pointer in the graph be `null` or point to a new node, thus ensuring that there can be no sharing. However, this simple strategy does not work; it creates trees, but we cannot easily control their size.

Figure 5-1 shows a generator adapted from an example for the QuickCheck tool [19] for random testing of Haskell programs. The `genSearchTree` method attempts to generate a random tree. It invokes `genNode` to generate a linked graph of nodes, and

```
SearchTree genSearchTree() {
    SearchTree tree = new SearchTree();
    tree.root = genNode();
    tree.size = tree.numNodes(tree.root);
    return tree;
}

Node genNode() {
    if (chooseBoolean()) {
        return null;
    } else {
        Node node = new Node();
        node.left = genNode();
        node.right = genNode();
        return node;
    }
}
```

Figure 5-1: A noncontrollable generator for trees.

then sets `size` to the number of generated nodes. Suppose that `chooseBoolean` returns a value with uniform probability: `true` with some probability p, and `false` with the probability $1 - p$. If $p < 0.5$, a call to `genNode` has less than 50% chance to terminate, because for `genNode` to terminate, both recursive calls must terminate. If $p > 2/3$, the expected size of the generated trees is less than one. Only carefully chosen values for $0.5 < p < 2/3$ enable the generation to terminate with nontrivial trees but can sometimes result in very large trees. To get a better control of the generated trees, we need to use the number of nodes during the generation, instead of computing it at the end.

Figure 5-2 shows a generator that passes the size to recursive calls. This generator can generate a random binary search tree, not just a tree, in a given `scope`. The `genSearchTree` method first chooses a value for the `size`. Suppose that the call `chooseInteger(lo, hi)` returns a random number between `lo` and `hi`, inclusive, if `lo` is not greater than `hi`; otherwise, the call fails, for example, throws an exception. The inputs to `genNode` are the size of the tree to generate and the bounds for the elements in the tree. The method randomly splits the size to the left and right subtrees, ensuring that the overall tree has the correct number of nodes.

The `genNode` method also chooses a value to put in the current node and appropriately adjusts the bounds for the subtrees. However, randomly choosing `info` to be between `min` and `max` can make the generation fail later on; the range of values may become smaller than the number of nodes in the tree, and eventually `min` becomes greater than `max` for some nonempty tree. (Since the tree implements a set, all elements should be different.) We can make the generation always succeed by changing the choice to: `chooseInteger(min + split, max - (size - 1 - split))`.

This example illustrates that developing even a random generator for relatively simple properties requires careful handling of intricate details. Developing further a

```
SearchTree genSearchTree(int scope) {
    int size = chooseInteger(0, scope);
    SearchTree tree = new SearchTree();
    tree.size = size;
    tree.root = genNode(size, 1, scope);
    return tree;
}

Node genNode(int size, int min, int max) {
    if (size == 0) {
        return null;
    } else {
        int split = chooseInteger(0, size - 1);
        int info = chooseInteger(min, max);
        Node node = new Node();
        node.info = info;
        node.left = genNode(split, min, info - 1);
        node.right = genNode(size - 1 - split, info + 1, max);
        return node;
    }
}
```

Figure 5-2: A controllable generator for binary search trees.

generator that can actually generate all structures within a scope requires more work. In a standard programming language such as Java, we would need to implement some iteration over the possible choices in chooseInteger. This in turn requires that generators return collections of substructures, merge them into larger structures, and potentially clone them before merging. Essentially, we would implement backtracking over possible choices.

We could instead use a language that has a built-in support for backtracking, for example, SETL [95], Icon [41], or AsmL [33]. In such language, we could use the same generator from Figure 5-2 to generate all structures within a scope, by treating each call to chooseInteger as a choice point for backtracking and appropriately changing the top-level calls to genSearchTree. Successive calls to genSearchTree would then systematically generate binary search trees. With backtracking, we may be even sloppier about the choice of values, allowing the generation to fail, for example, by choosing info as in Figure 5-2. Failures would only require backtracking to spend additional time exploring some invalid choices before generating a valid structure.

5.2 Comparison with Imperative Predicates

We next discuss the tradeoff in using generators and imperative predicates in bounded-exhaustive testing. We first compare the efficiency of generating all nonisomorphic test inputs within a scope using generators versus solving imperative predicates. We then compare the testing tasks for which we can use generators and imperative predicates. We finally compare how easy it is to develop generators and imperative predicates for the same property. Although generators generate test inputs faster, the other criteria make imperative predicates more appealing for bounded-exhaustive testing.

Generators that do not fail during backtracking, i.e., generate a valid structure for every invocation, can be much faster than solvers for imperative predicates. There are two reasons for this. First, solvers such as Korat necessarily fail during backtracking, i.e., explore some invalid candidate structures, and thus need to execute predicates several times before generating a valid structure. Second, generators do not have any execution overhead, whereas solvers such as Korat execute predicates with the overhead for monitoring the accessed fields. On the other hand, generators that fail during backtracking are not necessarily faster than solvers for imperative predicates.

Besides automatically generating test inputs, we also want to have automatic oracles that check correctness of the code under test for these inputs. We cannot practically use generators as partial oracles, but we can use imperative predicates. Suppose, for example, that we want to check whether the `remove` method preserves the binary search tree invariant checked with the `repOk` predicate. We can simply run `repOk` on the output object graph to check if it is a binary search tree. We could also hypothetically use a generator as an oracle: run the generator, increasing the scope, and check if it generates a tree that is isomorphic to the output object graph. It is clear, however, that this is very inefficient.

The main issue in comparing generators and predicates is how easy we can write them. For simple properties, such as binary search trees, it may be equally easy to write generators and predicates, but for more complex properties, it can be often easier to write predicates. Generators are particularly hard to write for properties that cannot be checked locally on a part of the structure. For example, a property of red-black trees requires every path from the root of the tree to a leaf to have the same number of black nodes [23]. Ball et al. [5] devoted a whole paper to the development of an efficient, specialized generator for red-black trees and its use in testing. In contrast, our experience shows that even undergraduate students can develop a predicate for red-black trees within a day of work. Tools such as QuickCheck [19] provide a library for writing generators for random generation, and languages that have backtracking can turn random generators into exhaustive generators, but the burden of implementing a specialized generator for each property is still on the programmer.

There are at least three more reasons why imperative predicates are easier to write than generators. First, generators are never present in the code, whereas some predicates are, usually in assertions or imperative specifications. Second, the programmer needs to ensure that generators avoid isomorphic structures—which is trivial for trees that have no sharing, but not trivial for structures that do have sharing—whereas solvers such as Korat automatically avoid isomorphic structures. Third, if we change the property, making it more specific—for example, instead of arbitrary trees, generate nonempty trees or complete trees—the programmer needs to rewrite an existing generator, while for an existing predicate, it is enough to add a new predicate that checks the additional property. We believe that programmers should consider a specialized generator for some property only when solving a predicate for that property is not efficient enough due to failures in backtracking.

5.3 Dedicated Generators

Dedicated generators are a Korat extension inspired by generators. We have found certain properties to be common in imperative predicates that describe class invariants (repOk predicates), for example, that a linked data structure is acyclic along some fields or that an array has all elements different or ordered. Similar properties are also present in combinatorial optimization problems and supported in libraries or languages such as Comet [78].

We have extended Korat with a library of dedicated generators that make it easier to write imperative predicates and also enable faster generation of valid structures. The library provides generators for the common checks. Each generator looks like an ordinary Java predicate and behaves like an ordinary Java predicate during regular execution. Imperative predicates, or any other code, can invoke these generators to check properties. However, during the generation of valid structures, Korat uses the special knowledge about the generators to further optimize its search.

For example, recall the repOk method for SearchTree from Figure 2-1. This method invokes isTree to check that the nodes reachable from the root field form a tree along the left and right fields. Appendix A shows how the programmer can write isTree in about 20 lines of Java code. Instead, the programmer could just use the library generator korat.isTree(root, new String[]{"left", "right"}). The generator is parameterized over the root node and the names of the fields. Given a root node, korat.isTree checks that the reachable nodes form a tree, i.e., no node repeats in the traversal of the nodes reachable from the root. The implementation of the generator takes into account this fact during the search.

When Korat generates a structure that satisfies korat.isTree along some fields, it does not try all (nonisomorphic) possibilities for those fields. Instead, each field is either null or points to a node that is not already in the tree. For the example finitization finSearchTree(s) from Figure 3-6, this reduces the number of possible values for a field from s+1 to 2. In the library, the implementation of korat.isTree uses the basic dedicated generator korat.isIn(field, set) that, while searching, assigns to the field only the values from the set, and while checking, checks that the value of field is in the set.

The library includes the *basic* dedicated generators for checking the following properties: (1) a value is in a set; (2) two values are equal; (3) a value is less/greater than another value; and (4) a value is of a certain class (instanceof). The library also includes *combinators* that allow creating complex generators for checking: (1) negation, (2) conjunction, and (3) disjunction. Finally, the library includes several higher-level generators, implemented using basic generators and combinators, which check structural constraints such as acyclicity or that elements of an array are sorted.

It is easy to add new dedicated generators to the Korat library; in theory, we could even add for each data structure a special-purpose generator that generates all valid structures without any failures in backtracking. However, we do not do that; the library that we use in the experiments presented in Chapter 8 has only generators that are applicable for several data structures. In practice, we do not expect Korat users to extend the library, but instead to use Korat as a general-purpose search.

Chapter 6

Testing Based on Imperative Predicates

This chapter presents how to apply solvers for imperative predicates in testing. The presentation uses our solver, Korat, but the concepts generalize to any other solver. Section 6.1 first presents specification-based, unit testing of Java methods, and specifically how Korat focuses on testing one Java method at a time. It then presents how to use Korat in testing sequences of method calls. Section 6.2 discusses that Korat is not restricted to unit testing and presents an example use of Korat in system testing.

6.1 Unit Testing

Unit testing is the process of testing each basic component of a program, i.e., a unit, to validate that it correctly implements its specification [120]. Unit testing is gaining importance with the wider adoption of Extreme Programming [8]. In unit testing, developers isolate the unit to run independently from its environment, which allows writing small testing code that exercises the unit alone. For Java programs, the basic testing unit can be either a method or a class.

We first give a brief overview of a widely used language for writing specifications for Java methods and classes. We then focus on testing a method as a unit and describe how Korat can automate test-input generation and correctness checking in this setting. We show how Korat can test whether methods implement their specifications by constructing and solving appropriate imperative predicates. In black-box testing, these predicates consist only of specifications, whereas in the white-box testing, the predicates consist also of the code under test. We finally illustrate how to use Korat in testing sequences of method calls.

6.1.1 Overview of Java Modeling Language

We next present some parts of the Java Modeling Language (JML) [67], a popular language for writing specifications for Java code. We introduce only the parts of JML that are relevant for testing based on Korat. More details are available elsewhere [67].

```
class SearchTree {
    //@ invariant repOk(); // class invariant
    ...
    /*@ normal_behavior // nonexceptional specification
      @     // precondition
      @     requires true;
      @     // postcondition
      @     ensures !contains(info) && \result == \old(contains(info));
      @*/
    boolean remove(int info) {
        ... // method body
    }
}
```

Figure 6-1: Example JML specification for the remove method.

Programmers can use JML to write specifications for methods and classes. Each method specification has a precondition, which specifies the property that should hold before the method invocation, and a postcondition, which specifies the property that the invocation should establish. Each class has a class invariant, which specifies the property that the methods should preserve: class invariant is implicitly both in the precondition and in the postcondition for each method. (More precisely, class invariants apply only to the public methods, but for simplicity our examples do not contain any Java visibility modifiers, and we assume that all members are public.)

Each method precondition, method postcondition, or class invariant expresses a property using a boolean expression. JML essentially uses Java syntax and semantics for expressions, with some extensions such as quantifiers. The JML tool-set can translate a large subset of JML into imperative Java predicates [17]; one of the most common uses of JML specifications is indeed to translate them into Java run-time assertions [14]. Programmers can thus use JML, and Korat based on JML specifications, without having to learn a specification language much different than Java. Moreover, since JML specifications can call Java methods, programmers can use the full expressiveness of the Java language to write specifications.

We next illustrate JML specifications using the examples from Chapter 2. Recall the methods `remove` for `SearchTree` and `extractMax` for `HeapArray` and their informal specifications from Figure 2-1 and Figure 2-4, respectively.

Figure 6-1 shows the JML annotations that specify partial correctness for the `remove` method from the `SearchTree` class. The JML keyword `normal_behavior` specifies that the method execution must return without throwing an exception if the precondition (specified with `requires`) holds right before the execution; additionally, the execution must satisfy the postcondition (specified with `ensures`).

In this example, the method `contains` returns true iff the tree contains the given element. The JML keyword `\result` denotes the return value of the method; `remove` returns `true` iff it removes an element from the tree. The JML keyword `\old` denotes that its expression should be evaluated in the pre-state, i.e., the state immediately before the method's invocation. The JML postcondition thus formally expresses the

```
class HeapArray {
    //@ invariant repOk(); // class invariant
    ...
    /*@ normal_behavior
      @     requires size > 0;
      @     ensures \result == \old(array[0]);
      @ also exceptional_behavior
      @     requires size == 0;
      @     signals (IllegalArgumentException e) true;
      @*/
    Comparable extractMax() {
        ... // method body
    }
}
```

Figure 6-2: Example JML specification for the `extractMax` method.

informal postcondition from Figure 2-1. The postcondition could also be stronger (e.g., that `remove` does not remove any element but `info`) or weaker (e.g., only that the method does not throw an exception); as we explain in Section 6.1.2, the postcondition does not affect the use of Korat for generating test inputs for `remove`.

We point out that the semantics of JML implicitly conjoins the class invariant (specified with `invariant`) with the precondition and postcondition. Thus, the full precondition for `remove` is `repOk() && true`; it requires that the input satisfy the `repOk` predicate. Similarly, the full postcondition is `repOk() && !contains(info) && \result == \old(contains(info))`; it requires that the output satisfy the `repOk` predicate and contain no `info`, and that the result has the appropriate value.

Figure 6-2 shows the JML annotations that specify partial correctness for the example `extractMax` method from the `HeapArray` class. This example specifies two behaviors. The `normal_behavior` specifies that whenever the input heap is valid and nonempty, the method returns the largest element from the original heap—without throwing an exception—and the resulting heap after the method execution is also valid. (The `repOk` predicate, implicitly conjoined to the preconditions and postcondition, checks the validity.)

JML can also express exceptional behaviors in which the method throws an exception. The example `exceptional_behavior` specifies that whenever the input heap is valid and empty, the method throws an `IllegalArgumentException`. The keyword `signals` specifies which exceptions the method throws and under which conditions; in this example, the condition `true` specifies that the method always throws an `IllegalArgumentException`. The rest of this section shows how to use Korat to check both normal and exceptional behaviors.

6.1.2 Black-Box Method Testing

We next present how to use Korat for specification-based, black-box testing of methods. In black-box testing, generation of test inputs considers only the specification

```
class SearchTree_remove {  // inputs to "remove"
    SearchTree This;       // implicit "this" argument
    int info;              // "info" argument

    // for black-box testing of "remove"
    boolean removePre() {  // precondition for "remove"
        return This.repOk();
    }
}
```

Figure 6-3: Predicate for black-box testing of the `remove` method.

of the code under test, and not the code itself. Valid test inputs for a method must satisfy its precondition. Based on Korat and the JML tool-set [17], we have built a testing tool that generates valid test inputs for methods and checks their correctness. The tool uses bounded-exhaustive testing: Korat generates all nonisomorphic structures (within a scope) that satisfy the precondition, and the tool executes the method on each input and checks the output using the postcondition as a test oracle.

Generating Test Inputs

To generate test inputs for a method *m*, our tool first constructs a Java class that corresponds to the method inputs. This class has one field for each method argument, including the implicit `this` argument. The class also has a predicate that corresponds to the method precondition. The tool then uses Korat, with an appropriate finitization, to generate all nonisomorphic valid structures for the new predicate; each of these structures corresponds to a valid test input for the method *m*.

Figure 6-3 shows the class for the inputs to the example `remove` method. Each input is a pair of a `SearchTree` object (the field `This` represents the implicit argument `this`) and an integer (the field `info` represents the argument `info`). The predicate `removePre` checks the precondition for `remove`: it invokes `repOk` on the implicit `this` argument and leaves `info` unconstrained. Figure 6-4 shows the finitization skeleton that Korat creates for the new class, reusing the finitization for `SearchTree`, and a specialized version of this skeleton for a given `scope`.

For the example `extractMax` method, the specification describes two behaviors. We can choose to separately generate inputs that satisfy each of the preconditions, or we can choose to generate all inputs at once: the predicate is then a disjunction of the preconditions for different behaviors.

Checking Output Correctness

After generating all valid test inputs for a method, our testing tool invokes the method on each input and checks each output with a test oracle. A simple test oracle could just reuse the invariant: to check partial correctness of the method, invoke `repOk` in the post-state to check if the method preserves its class invariant. If the result is `false`,

```
Finitization finSearchTree_remove(int numNode,
        int minSize, int maxSize, int minInfo1, int maxInfo1,
        int minInfo2, int maxInfo2) {
    Finitization f = new Finitization(SearchTree_remove.class);
    Finitization g = finSearchTree(numNode, minSize, maxSize,
                                   minInfo1, maxInfo1);
    f.includeFinitization(g);
    f.set("This", g.getRootObject());
    f.set("info", new IntSet(minInfo2, maxInfo2));
    return f;
}

Finitization finSearchTree_remove(int scope) {
  return finSearchTree_remove(scope, 0, scope, 1, scope, 1, scope);
}
```

Figure 6-4: Finitizations for the precondition of the remove method.

the method under test is incorrect, and the input provides a concrete counterexample. Programmers can also manually develop more elaborate test oracles.

Our current implementation automatically generates test oracles from method postconditions (and other assertions) using a slightly modified JML tool-set based on the JMLunit framework [17]. The JML tool-set translates JML postconditions into runtime Java assertions. If an execution of a method violates such an assertion, the method throws an exception to indicate a violated postcondition. Test oracle catches these exceptions and reports correctness violations. These exceptions are different from the exceptions that the method specification allows, and our tool uses JML to check both normal and exceptional behavior of methods. More details of the JML tool-set and translation are available elsewhere [67]. We point out that the oracles based on JML support assertions that can appear anywhere in the code, not only postconditions that appear at the end of methods.

Our tool also uses JMLunit to combine JML test oracles with JUnit [9], a popular framework for unit testing of Java classes. JUnit automates test execution and error reporting, but requires programmers to provide test inputs and test oracles. JMLunit further automates both test execution and correctness checking. However, JMLunit requires programmers to provide sets of possibilities for all method arguments (some of which can be complex structures) and generates all valid inputs by generating *all* tuples from the Cartesian product of the possibilities and filtering the tuples using preconditions. Korat, on the other hand, constructs structures from a simple description of the fields in the structures and additionally does not try all possible tuples but (1) prunes the search based on the accessed fields and (2) generates only one representative from each isomorphism partition. The use of Korat thus automates generation of test inputs, automating the entire testing process. Figure 6-5 summarizes the comparison of these testing frameworks.

	Testing framework		
testing activity	JUnit	JMLunit	Korat+JMLunit
generating test inputs			√
generating test oracle		√	√
running tests	√	√	√

Figure 6-5: Comparison of several testing frameworks for Java. The mark "√" indicates the testing activities that the frameworks automate.

6.1.3 White-Box Method Testing

We next present how to use Korat for specification-based, white-box testing of methods. In white-box testing, the predicate that Korat solves includes both the specification and the code of the method under test. Korat then monitors the execution of the code under test, which allows Korat to prune the search for inputs when the code does not depend on a part of the input. To test a method, we first construct a predicate that corresponds to the negation of the method's correctness. If Korat finds a valid input for this predicate, the method is incorrect, and the input provides a counterexample.

Figure 6-6 shows the predicate `removeFail` for the example `remove` method. This predicate first executes `removePre`: if it returns `false`, the input is not a valid test input for `remove` and cannot be a counterexample; otherwise, the input is valid, and the predicate executes `remove`, together with the JML-translated assertions. If this execution throws a JML exception, `remove` failed to satisfy its specification.

Comparison with Black-Box Method Testing

The difference between predicates for white-box and black-box testing is in the invocation of the method under test; in our example, `removeFail` invokes `remove`, whereas `removePre` does not. It means that to generate valid inputs for `removeFail`, Korat instruments `remove`, among other methods, and monitors the accesses that `remove` makes to the candidate. This by itself makes one execution of `remove` slower. However, it "opens" the body of `remove` for the optimizations that Korat performs to prune the search. In situations where the method under test and its assertions do not depend on the whole input, this can significantly reduce the time to test the method. Our experiments in Chapter 8 use postconditions that traverse the whole reachable object graph, and thus do not benefit from white-box testing.

6.1.4 Sequences of Method Calls

We have so far presented how to use Korat for testing one method at a time; we next consider testing sequences of method calls. It is straightforward to translate the problem of testing a fixed sequence to the problem of testing one method and then

```
class SearchTree_remove {  // inputs to "remove"
    SearchTree This;       // implicit "this" argument
    int info;              // "info" argument

    // for white-box testing of "remove"
    boolean removeFail() { // failure for "remove"
        if (!removePre()) return false;
        try { // invoke "remove" with JML assertions
            This.remove(info);
        } catch (JMLAssertionException e) {
            return true;   // assertion not satisfied
        }
        return false;
    }
}
```

Figure 6-6: Predicate for white-box testing of the `remove` method.

apply Korat either in black-box or white-box setting. We also discuss how to use Korat to generate sequences of method calls.

Sequences often arise in the context of algebraic specifications [43], for example an axiom specifying that an insertion of an element in a container followed immediately by a deletion of the element leaves the container unchanged. This example illustrates a fixed sequence specifying a property of `remove`: `t.add(e).remove(e).equals(t)`. Sequences can also describe correctness properties at an abstract level, as in model-based specifications [100] using abstraction functions [70]. For example, the following sequence specifies a model-based property for `remove`:

```
JMLObjectSet sPre = t.abstract();
t.remove(e);
JMLObjectSet sPost = t.abstract();
return sPost.equals(sPre.delete(e));
```

The sequence abstracts each tree into a set; removing an element from a tree commutes with removing the element from the set. (We can express this form of specification in JML using `model` fields [67].) Figure 6-7 shows the methods `axiom` and `implements` that correspond to the two example sequences. We can use Korat to test these methods the same way as to test `remove`.

Korat not only enables testing fixed sequences of method calls but also enables *generating* sequences of interest. The user needs to build a representation of desired sequences and a predicate that defines their validity; solving the predicate then provides sequences with desired property. Korat can generate all such sequences up to a given length. Each sequence corresponds to a test input that consists of method calls; this contrasts the typical use of Korat to generate inputs as concrete structures. Generation of sequences also allows Korat to discover a sequence that builds a given structure such that Korat can output counterexamples, or other inputs of interest, as sequences of method calls.

```
/*@ requires t.repOk();
  @ ensures \result == true; */
boolean axiom(SearchTree t, Comparable e) {
    return t.add(e).remove(e).equals(t);
}

/*@ requires t.repOk();
  @ ensures \result == true; */
boolean implements(SearchTree t, Comparable e) {
    JMLObjectSet sPre = t.abstract();
    t.remove(e);
    JMLObjectSet sPost = t.abstract();
    return sPost.equals(sPre.delete(e));
}
```

Figure 6-7: Methods corresponding to the example method sequences.

6.2 System Testing

Solvers for imperative predicates are not restricted to unit testing. We briefly present a case study that used a solver for imperative predicates for system testing of a real-world application. According to the traditional testing process [10], after unit testing, which tests each unit in isolation, developers perform integration testing, which tests interactions between several units, and system testing, which tests the whole system. For software such as command-line programs or interactive applications, it is easy to draw a boundary between units and systems: we can use a system, but not units, stand-alone. However, for modern software that involves "middleware", "components", "frameworks" etc., it is harder to draw a boundary: one user's unit may be another user's system.

We consider a system to be a piece of software whose precondition is `true`, i.e., a system is able to deal with any input, e.g., a compiler should process a file with any sequence of characters. This typically means that a system has a huge potential number of inputs. Many of these inputs may be illegal in that the system only reports an error for such inputs, e.g., most sequences of characters are not syntactically or semantically well-formed programs. Thus, a main issue in system testing is to generate test inputs that consider one aspect of the system. Our approach is to let the user describe the properties of system test inputs of interest with an imperative predicate and then use a solver to generate such inputs. Another issue is to provide an oracle that checks the correctness of code for the automatically generated inputs. In this setting, the predicate for test inputs cannot be an oracle. We next describe how differential testing provides an automatic oracle.

6.2.1 Differential Testing

Differential testing, also known as n-version testing, is a testing technique that uses one implementation of an algorithm as an oracle for checking another implementation

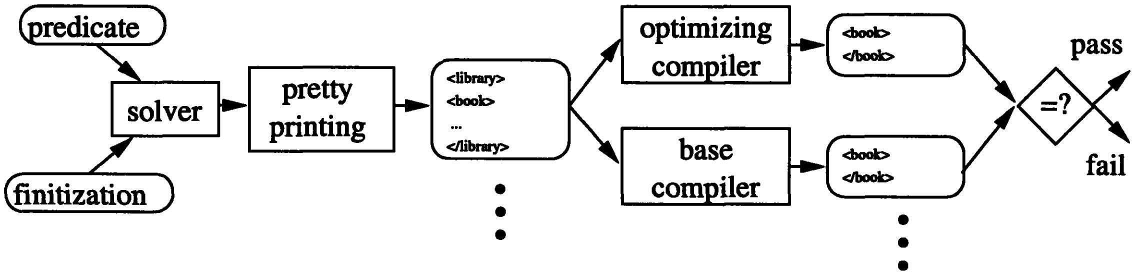

Figure 6-8: Differential testing of an XPath compiler.

of the same algorithm [76]: we run both, or in general n, implementations on the same input, and if their outputs differ, the input potentially detects a fault. A common use of differential testing is in regression testing; after modifying a program, the developer establishes that the original version and the modified version generate the same output for a range of inputs.

We next describe a realistic example that illustrates the use of solvers for imperative predicates and differential testing in detecting faults in production-quality code. The Foundations of Software Engineering group at Microsoft Research implemented a solver for imperative predicates as a part of the AsmL Test Tool (AsmLT) [33]. We present more details of the AsmLT in Chapter 10; suffice to say that the solver is for predicates written in the AsmL language [42], not Java or C#. Testers at Microsoft have used AsmLT to detect faults in several systems, including an optimizing XPath compiler [102]. XPath is a language for querying XML documents [115]: we can view an XPath compiler as taking an XPath query and an XML document and producing another XML document. (More precisely, an XPath compiler takes an XPath query and produces a program that processes XML documents.)

The goal was to test the optimizing version of the XPath compiler; there existed a base version that processed the XPath queries but did not optimize them. The base compiler was smaller and more mature, so the confidence in its correctness was high. The testers used differential testing to check the optimizing compiler: they obtained a set of test inputs, consisting of XML documents, then ran the inputs through both compilers to compare the outputs. The testers generated the test suite manually and augmented it with XML documents randomly generated from a grammar [75, 97]. Several faults were detected in the optimizations and corrected; the testers believed that the optimizing compiler was relatively free of faults. It was well-tested, according to the best testing practices, and it was about to be released. However, the testers then used AsmLT to automatically generate more inputs with specific properties and detected ten more faults in the optimizing compiler.

Figure 6-8 shows the use of a solver for imperative predicates and differential testing in checking the optimizing XPath compiler:

1. The testers created a *model* of test inputs, namely, XML syntax trees. The input to the compiler is essentially a sequence of characters, but it was easier to

express the semantic properties of the desired inputs at the level of the syntax trees than directly as a sequence of characters. Also, the use of syntax trees ensures that the test inputs are syntactically correct XML documents. Note that the testers wanted to test particular compiler optimizations, not parsing or error reporting.

2. The testers wrote a predicate that expressed the property of the desired test inputs, namely, which XML elements should appear in the syntax tree and in what relationships.

3. From a predicate and a finitization, the solver generates structures that satisfy the predicate, as usual. These structures are XML syntax trees, so it was necessary to pretty-print them to obtain the actual test inputs as XML documents.

4. A tool then runs each test input through both versions of the code and compares the outputs. When they differ, it is necessary to find which version actually has a fault. In this particular application, all faults were in the optimizing compiler.

In conclusion, the use of a solver for imperative predicates enabled testers to detect unknown faults in already well-tested, production-quality code.

Chapter 7

Mutation Testing

This chapter presents Ferastrau, our tool for mutation testing of Java programs. Section 7.1 gives a brief overview of mutation testing and its uses. Section 7.2 presents the design and implementation of Ferastrau. We used Ferastrau to evaluate the test suites that Korat generates for bounded-exhaustive testing; Chapter 8 presents the evaluation.

7.1 Overview

Mutation testing [28,44,86] is a testing technique based on fault injection. The main assumption underlying mutation testing is that a test suite that detects many injected faults can also detect real faults. We first present how to perform mutation testing and then discuss the strength of this technique.

A tool for mutation testing proceeds in two steps. In the first step, the tool generates a set of *mutants* by modifying the original program. The tool applies *mutation operators* that perform small syntactic modifications on the original program. For example, a modification can replace a variable with another variable (of a compatible type), say `n.left` with `n.right`. Mutation operators model typical small software errors that developers make. The literature contains operators for several programming languages [2,62,87], including Java [59,60,71]. The operators for Java include the traditional operators that modify statements and expressions, as well as object-oriented operators that modify classes, for example, field or method declarations.

In the second step, the tool measures how many mutants a given test suite detects: the tool executes each mutant and the original program on each test input from the suite and compares the corresponding outputs. If a mutant generates an output different from the original program, we say that the test input *kills* the mutant. The *mutation coverage* is the ratio of the number of killed mutants to the total number of mutants; the higher the coverage, the better the test suite is according to this metric.

Mutation criterion requires a test suite to kill all (nonequivalent) mutants. Some of the mutants may be semantically equivalent to the original program, although syntactically different, so there is no input that can kill them. Since determining program equivalence is undecidable in general, the practical goal is to kill almost all (nonequivalent) mutants. A test suite is *mutation-adequate* if it kills a prespecified percentage of mutants. We can eliminate from the mutant pool all mutants which we find, manually or automatically, to be equivalent to the original program.

As mentioned, the main assumption of mutation testing is that a mutation-adequate test suite is very likely to detect the actual faults in the original program [86]. In other words, some test input that kills a mutant also makes the original program generate an incorrect output. It is up to the user to establish that the output is incorrect, either manually or using an oracle that automatically checks the (partial) correctness. Numerous studies have used mutation testing for measuring the quality of test suites; for a review see [86]. The experimental results showed that mutation criterion was stronger than testing criteria based on code coverage [111] and data flow [34]. Thus, evaluations of testing approaches often use mutation testing [16], and we also chose mutation testing for evaluation of bounded-exhaustive testing.

7.1.1 Output Comparison

We have so far used "output comparison" with the intuitive meaning. We next discuss how to compare outputs of Java methods in mutation testing. We use a strong criterion for output comparison, which affects the whole design of Ferastrau, a tool that we have developed for mutation testing of Java programs. Recall that an output of a Java method consists of the return value and the objects in the post-state (Section 2.1). Comparison of primitive values is straightforward; however, the objects can represent complex structures, and it is not clear a priori how to compare them.

Traditional mutation testing compares the whole post-state with equality. For Java states, however, comparison of the object graphs with equality is a weak criterion: because of object identities, this comparison would find almost all states to differ and thus kill almost all mutants. We need to compare Java states using at least the isomorphism of object graphs. Ferastrau, however, uses an even stronger criterion.

Ferastrau allows the user to provide a binary predicate—a method that takes two inputs and returns a boolean value—for comparison of complex structures. By default, Ferastrau uses the `equals` methods, following the Java convention of using `equals` for comparison of objects. This allows comparisons based on *abstract* values; for example, two binary search trees that implement sets may be structurally different at the *concrete* level of the implementation, but if they represent the same set, they are equal according to the `equals` method. For most benchmarks, the existing `equals` method suffices for comparing outputs, but the user may provide another method. For example, the user could deem two trees to be "equal" if they have the same structure, irrespective of the elements. The use of `equals` requires Ferastrau to generate mutant classes that have the same name as the corresponding original classes.

7.1.2 Existing Tools

We next review the existing mutation-testing tools. Mothra [62] and Proteum [26] are two most popular tools for mutation testing of Fortran and C programs, respectively. Researchers have recently developed several tools for Java [15,18,81,85] in addition to our tool, Ferastrau. All tools for Java, including Ferastrau, have conceptually similar mutant generation, although the actual mutation operators used and the implementation approaches may differ. Where Ferastrau really differs from the other tools is in output comparison and mutant execution.

Chevalley and Thévenod-Fosse [18] present a tool that generates mutant for Java but provides no mutant execution. Jester [81] and MU [15] are two tools that both generate mutants and provide execution through JUnit [9], a popular framework for unit testing. These tools, however, do not compare the states; they only compare the boolean results from the JUnit execution of a test, namely, pass or fail.

MuJava [71,85] is a mutation-testing tool for Java that requires the user to write each test input such that it outputs a `String` object. MuJava then compares the outputs by comparing these strings. This approach requires the user to provide a specific comparison for each test input, whereas Ferastrau allows the user to provide one method that can compare outputs for all test inputs. MuJava supports mutant execution but does not handle nontermination and exceptions; Section 7.2.2 presents how Ferastrau handles nontermination and exceptions.

7.2 Ferastrau: A Mutation Tool for Java

We next present the implementation of Ferastrau. We first present how Ferastrau generates mutants. We then present how Ferastrau executes mutants and compares outputs. Ferastrau handles all possible outcomes in the execution of Java mutants, including nontermination and exceptions.

7.2.1 Mutant Generation

We implemented mutant generation in Ferastrau by changing the Sun's `javac` compiler to perform a source-to-source translation: Ferastrau parses each class of the original program into an abstract syntax tree, applies the mutation operators to the trees, and outputs the source of the mutants. Ferastrau uses the following mutation operators:

- Replace a Java operator with another operator (of the same type), e.g., '+' to '-', '==' to '`!=`', '<' to '<=' etc.

- Replace a variable with another variable (of a compatible type), e.g., a local variable `i` to `j` or an instance variable `n.left` to `n.right`.

- Replace an invocation of a method with another method (of a compatible signature). Ferastrau does not replace some special methods, such as `notify`.

These operators modify only the code of methods, and not the classes. They correspond to the traditional mutation operators for procedural languages [2, 62] and can create subtle mutants that only nontrivial inputs can kill, as the results in Section 8.3.2 show. It is easy to add new operators to Ferastrau such as the operators proposed for class mutation [59,60,71], but these operators are not necessary for the benchmarks that we use in Chapter 8.

Ferastrau allows the user to provide bounds for the number of generated mutants. Ferastrau first computes the possible number of mutations (i.e., the number of applications of mutation operators) for each class in the original program. It next computes the probabilities with which to apply each mutation to generate a number of mutants between the bounds. It then randomly selects the mutations to apply to generate each mutant. Traditionally, mutation-testing criterion requires that a tool generates one mutant for each possible mutation, which usually results in a large number of mutants and presents a big challenge for the performance of mutation-testing tools. However, the results show that randomly selecting only a certain portion of mutants still provides an effective mutant pool [113].

For each original class, Ferastrau generates mutants with the same class name. For reasons explained below, Ferastrau provides two approaches: (1) for each mutant, generate one class that has both the code for the original class and the mutant (and additionally a flag that determines which code to execute) or (2) generate mutants as separate classes. Suppose that the original code contains `temp.right` that is to be mutated to `left.right`. The first approach uses *metamutants* [106]: it guards the mutations by boolean variables that are appropriately set during the mutant execution; it generates one class with `(MUTANT ? left :  temp).right`. The second approach simply generates `left.right` in a separate class.

7.2.2 Mutant Execution

After generating the mutants, Ferastrau runs them on a set of test inputs, manually or automatically generated. Ferastrau executes the original code and the mutants for each input and compares their respective outputs. We had two requirements for Ferastrau:

- Ferastrau had to handle a potentially huge number of relatively small inputs to be practical for an evaluation of bounded-exhaustive test suites that Korat generates (Chapter 8). The question then is whether Ferastrau should execute the original code and the mutants in a single run or in separate runs?

- Ferastrau had to handle all outcomes of Java executions. We have discussed in Section 7.1.1 how to compare nonexceptional Java outputs, but the question is how to handle nontermination and exceptional termination of the original code and the mutants?

We next describe how Ferastrau addresses these questions and the rationale behind our decisions. We then list the conditions that Ferastrau uses to kill a mutant.

Ferastrau executes the original code and the mutants in a single run; otherwise, it would need to serialize all the outputs, which could take a long time and produce very large files for bounded-exhaustive test suites. Recall that Ferastrau can generate metamutants or generate the original code and the mutants in different classes. If Ferastrau uses metamutants, the guarding boolean variables slow down the execution.

We have found this approach to be faster for shorter executions and larger data. If Ferastrau generates different classes, it needs to execute several classes with the same name in a single Java Virtual Machine (JVM). Ferastrau then uses a different `ClassLoader` [105] for the original class and each mutant, and uses serialization through a buffer in memory to compare objects. We have found this approach to be faster for longer executions and smaller data.

Ferastrau assumes that the original code terminates for all test inputs, either normally or exceptionally.[1] If the specification allows an exceptional termination, it is not an error. Ferastrau handles nontermination of mutants by running them in a separate thread with a time limit for execution. The limit is set to $T_m = 10T_o + 1sec$, where T_o is the time the original code runs for that input. We have found these constants sufficient to account for fluctuations in the execution time of Java programs, e.g., due to garbage collection. In all examples that we tried, a mutant that did not finish in $10T_o$, did not finish in $30T_o$ either. The minimum of $1sec$ is necessary, at least for Sun's Java 2 SDK1.3.x JVMs, because JVMs take some extra time for the executions that raise exceptions.

The mutants can also terminate, either normally or exceptionally. Ferastrau catches all exceptions (in Java, all `Throwable` objects) that the executions throw. This allows Ferastrau to compare the outputs, even exceptional, as well as to catch all errors in the mutants. The errors include the cases when the mutant runs out of stack or heap memory and JVM raises `StackOverflowError` or `OutOfMemoryError`, respectively. Although the JVM specification [69] does not precisely specify the behavior when `OutOfMemoryError` is raised, we found several Sun's Java 2 SDK1.3.x JVMs to be able to continue the execution after the garbage collection of the objects allocated by the thread for the erroneous mutant.

Ferastrau uses the following conditions to kill a mutant for some test input:

- The mutant's output does not satisfy some class invariant (`repOk`), which is a precondition for `equals`.
- The mutant's output differs from the output of the original code; any of the outputs can be normal or exceptional.
- The mutant's execution exceeds the time limit.
- The mutant's execution runs out of memory.

Note that the executions of mutants do not check the (JML) postconditions; indeed, if all methods had complete functional specifications, we could simply check the postconditions to kill mutants and would not even need to compare outputs.

[1] The user can always put a time limit on the execution; if a test exceeds the limit, it potentially detected a fault that leads to nontermination of the original code.

Chapter 8

Evaluation

This chapter presents an evaluation of Korat and bounded-exhaustive testing. Section 8.1 discusses the benchmark suite that we use in the evaluation. The suite consists of ten data-structure implementations. Section 8.2 presents the performance of Korat in generating valid structures, with and without dedicated generators. Section 8.3 presents our evaluation of bounded-exhaustive testing of data structures.

Recall that bounded-exhaustive test suites consist of all nonisomorphic test inputs within a given scope. We evaluate the performance of generating bounded-exhaustive test suites and the quality of these suites. As presented in Section 1.2.1, we use code coverage and mutation coverage to assess the quality of test suites; we say that a scope is adequate for some criterion if the bounded-exhaustive test suite for that scope is adequate for that criterion. Specifically, we evaluate the following four hypotheses for our benchmarks:

- **Mutation:** There is a certain small mutation-adequate scope, i.e., bounded-exhaustive test suite for this scope can detect almost all nonequivalent mutants.

- **Coverage:** The mutation criterion is stronger than the code coverage criterion, i.e., mutation-adequate scope is larger than code-coverage adequate scope.

- **Feasibility:** Korat can effectively generate bounded-exhaustive test suites for the mutation-adequate scope, and it is practical to check correctness for these test suites, despite the potentially large number of test inputs.

- **Randomness:** Bounded-exhaustive test suites for a mutation-adequate scope can be more effective than random test suites with the same number of inputs randomly selected from a larger scope.

We performed all experiments on a Linux machine with a 1.8 GHz Pentium 4 processor using Sun's Java 2 SDK 1.3.1 JVM.

8.1 Benchmarks

Figure 8-1 lists the benchmarks that we use in experiments. The benchmarks include our implementations of text-book data structures, some data structures from the

benchmark	package	finitization parameters
`SearchTree`	`korat.examples`	numNode, minSize, maxSize, minInfo, maxInfo
`HeapArray`	`korat.examples`	minSize, maxSize, maxLength, maxElem
`BinHeap`	`korat.examples`	minSize, maxSize
`FibHeap`	`korat.examples`	minSize, maxSize
`DisjSet`	`korat.examples`	minSize, maxSize
`LinkedList`	`java.util`	minSize, maxSize, numEntries, numElems
`SortedList`	`java.util`	minSize, maxSize, numEntries, numElems
`TreeMap`	`java.util`	minSize, numEntries, maxKey, maxValue
`HashSet`	`java.util`	maxCapacity, maxCount, maxHash, loadFactor
`AVTree`	`ins.namespace`	numAVPairs, maxChild, numStrings

Figure 8-1: Benchmarks and finitization parameters. Each benchmark is named after the class for which data structures are generated; the structures also contain objects from other classes.

Java Collections Framework that is a part of the standard Java libraries [105], and intentional names from the Intentional Naming System [1]:

- `SearchTree` is our main running example, a binary search tree implementation.
- `HeapArray` is the other running example, an array-based implementation of the heap data structure. This benchmark is representative for array-based data structures such as stacks and queues, as well as `java.util.Vector`.
- `BinHeap` is a linked data structure that implements binomial heaps [23].
- `FibHeap` is a linked data structure that implements Fibonacci heaps, which have different structural invariants and complexity for some operations than binomial heaps [23].
- `DisjSet` is an array-based implementation of the fast union-find data structure [23]; this implementation uses both path compression and rank estimation heuristics to improve efficiency of find operations.
- `LinkedList` is an implementation of linked lists from the Java Collections Framework. This implementation uses doubly-linked, circular lists. This benchmark is also representative for linked data structures such as stacks and queues. The elements in `LinkedList` are arbitrary objects.
- `SortedList` is structurally identical to `LinkedList`, but has sorted elements.
- `TreeMap` implements the `Map` interface using red-black trees [23]. The implementation uses binary trees whose nodes have a `parent` field, as well as a `key` and a `value`. Testing `TreeMap` with all `value` fields set to `null` corresponds to testing the set implementation in `java.util.TreeSet`.

- HashSet implements the Set interface, backed by a hash table [23]. This implementation builds collision lists for buckets with the same hash code. The loadFactor parameter determines when to increase the size of the hash table and rehash the elements.

- AVTree implements the intentional name trees that describe services in the Intentional Naming System (INS) [1], an network architecture for service location. Each node in an intentional name has an attribute, a value, and several children nodes. INS uses attributes and values to classify services. INS represents these properties with arbitrary String objects except that "*" is a wildcard that matches all other values. The finitization bounds the number of AVPair (attribute-value pair) objects that implement nodes, the number of children for each node, and the total number of String objects (including the wildcard). The original implementation of INS had faults that we detected with bounded-exhaustive testing [73] and corrected. We use the corrected version as the original program in these experiments, but (most of) the mutants are faulty.

8.2 Generation of Data Structures

Figure 8-2 presents the results for generating valid structures for our benchmark suite. For each benchmark, we set all finitization parameters such that Korat generate structures of *exactly* the given size: we set minSize, maxSize, and other maximum parameters to the value of size and the minimum parameters to 0 or 1, according to the structure[1]. (For bounded-exhaustive testing, Korat generates all test inputs *up to* the given bound.) For a range of size values, we tabulate the number of connected structures in the state space (Section 3.2.3), the number of valid structures that Korat generates, and the number of candidate structures that repOk checks and the time that Korat takes to generate all nonisomorphic valid structures, without and with dedicated generators.

We can see that the state spaces are very large for these sizes. Hence, the naive algorithm that would try as candidates all structures from the state space (Section 3.1.3) would not terminate in a reasonable time. Further, the ratio of the number of valid structures and the size of state space shows that these state spaces are very sparse: for each valid structure, there is a large number of invalid structures. For example, for SearchTree of size seven, there are 2^{59} connected structures in the state space, and only 429 of them are valid nonisomorphic structures, i.e., about one in 10^{16} structures is a valid nonisomorphic structure.

The results show that Korat can efficiently generate all valid nonisomorphic structures even for very large state spaces. The search pruning that Korat performs allows Korat to explore only a tiny fraction of the state space. The ratio of the number of candidate structures considered and the size of the state space shows that the key to

[1] The only exception is the loadFactor parameter for HashSet, which is set to 1.00 for comparison with TestEra (Chapter 9). We previously showed the results for loadFactor being set to the default value of 0.75 [13].

				without dedicated		with dedicated	
benchmark	size	state	valid	candidates	time	candidates	time
		space	structures	considered	[sec]	considered	[sec]
SearchTree	7	2^{59}	429	340990	5.69	69355	1.48
	8	2^{71}	1430	2606968	48.54	475042	7.65
	9	2^{84}	4862	20086300	377.83	3312243	50.23
	10	2^{97}	16796	155455872	3480.63	23379912	413.13
HeapArray	6	2^{20}	13139	64533	0.72	17766	0.65
	7	2^{25}	117562	519968	2.03	150084	1.31
	8	2^{29}	1005075	5231385	15.30	1295739	9.33
	9	2^{34}	10391382	51460480	151.50	12964446	76.68
BinHeap	7	2^{103}	107416	307106	4.54	187542	2.60
	8	2^{125}	603744	1428740	20.82	921464	11.19
	9	2^{147}	8746120	13512131	229.90	10525015	146.13
FibHeap	4	2^{56}	2310	8634	0.76	6775	0.74
	5	2^{78}	52281	164365	2.53	134965	2.40
	6	2^{101}	1474186	4809294	61.29	3951101	61.05
DisjSet	4	2^{26}	914	7915	0.61	6677	0.60
	5	2^{37}	41546	413855	2.03	354015	1.82
	6	2^{49}	2967087	33436639	133.86	28864251	158.00
LinkedList	7	2^{71}	4140	4269	0.81	4186	0.66
	8	2^{84}	21147	21306	0.94	21199	0.84
	9	2^{97}	115975	116167	2.39	116033	2.03
	10	2^{111}	678570	678798	11.62	678634	9.21
SortedList	11	2^{125}	352716	3880154	58.99	705512	12.50
	12	2^{140}	1352078	16225257	264.72	2704243	49.82
	13	2^{155}	5200300	67604267	1154.23	10400694	225.52
TreeMap	7	2^{86}	35	67259	2.35	31102	1.46
	8	2^{104}	64	306486	8.78	153883	4.63
	9	2^{123}	122	1447664	44.25	797294	22.17
	10	2^{142}	260	7264732	242.81	4309282	118.87
HashSet	7	2^{104}	1716	100861	2.33	99580	1.80
	8	2^{126}	6435	441930	8.85	436932	5.65
	9	2^{148}	24310	1910842	38.99	1891402	24.69
	10	2^{172}	92378	8179746	177.47	8104173	109.51
AVTree	3	2^{30}	84	336	0.57	307	0.56
	4	2^{56}	5923	16118	0.89	15933	0.80
	5	2^{91}	598358	1330628	24.66	1329637	18.84

Figure 8-2: Performance of Korat for generation of data structures. All maximum finitization parameters are set to the size value. State size is rounded to the nearest smaller exponent of two. Time is the elapsed real time in seconds for the entire generation.

effective pruning is backtracking based on fields accessed during predicate executions. For example, for SearchTree of size seven, Korat explores 340,990 candidates out of 2^{59} possible structures in the state space, i.e., less than one candidate in 10^{12} structures. Without backtracking, and even with isomorphism optimization, Korat would generate infeasibly many candidates. Isomorphism optimization further reduces the number of candidates, but it mainly reduces the number of valid structures. For example, for SearchTree of size seven, isomorphism optimization reduces the number of candidates and structures $7! > 5000$ times. Without isomorphism optimization, Korat would generate infeasibly many valid structures.

For SearchTree, LinkedList, SortedList, TreeMap, and HashSet (with the parameter loadFactor set to 1.00), the numbers of nonisomorphic structures appear in the On-Line Encyclopedia of Integer Sequences [98]. For all sizes for these benchmarks, Korat generates the correct number of structures. We have manually inspected the structures generated for small sizes for other benchmarks. In all these cases, Korat correctly generated all valid nonisomorphic structures. These results increase our confidence in the correctness of the implementation of Korat; although we have proved the Korat algorithm correct (Chapter 4), we might still have errors in the implementation.

8.2.1 Dedicated Generators

Recall from Section 5.3 that Korat provides a library of dedicated generators for common properties. We evaluate the effectiveness of dedicated generators by comparing the performance of Korat with and without them. The results in Figure 8-2 show that the use of dedicated generators can speed up valid structure generation for up to 88% or almost 8.5 times (for SearchTree and size ten).

Dedicated generators make the generation faster because they reduce the number of candidates, pruning the search beyond the backtracking based on accessed fields. However, dedicated generators make each execution of a candidate somewhat slower because of (1) higher overhead for the execution of Korat library and (2) the time dedicated generators take to produce the next candidate. We can see that the use of dedicated generators may even slow down the generation (for DisjSet and size six). The reason is that DisjSet uses only one dedicated generator that makes all elements of an array different, which is simple in comparison to the rest of repOk; the benefit of this simple generator is thus outweighed by the increased run-time cost for the rest of repOk. In such situations, the user does not need to use the dedicated generator.

As mentioned in Section 5.3, Korat library does not aim to provide the most efficient generators for our benchmarks. We could in principle add for each data structure a special-purpose dedicated generator that would generate the structures directly without any failures in search, making the number of candidates the same as the number of valid structures and the generation even faster. However, the Korat library instead provides dedicated generators that are applicable for several data structures. These dedicates predicates also make it easier to write the actual predicates for all our benchmarks.

benchmark	target methods	some helper methods	ncnb lines	# br.	# mut.
SearchTree	add, remove	contains	85	20	272
HeapArray	insert extractMax	heapifyUp heapifyDown	51	9	274
BinHeap	insert, delete extractMin, union	contains, decrease merge, findMin	182	33	292
FibHeap	insert, delete extractMin, union	contains, decrease cascadingCut, cut consolidate	171	31	297
DisjSet	union, find	compressPath	29	8	243
LinkedList	add, remove reverse	contains ListIterator.next	102	16	244
SortedList	insert, remove sort, merge	contains	176	29	231
TreeMap	put, remove	get, containsKey fixAfterInsertion fixAfterDeletion rotate{Left,Right}	230	47	293
HashSet	add, remove	contains HashMap.containsKey HashMap.put HashMap.remove	113	20	244
AVTree	lookup	extract	199	26	205

Figure 8-3: Target methods for testing data structures. Korat generates inputs for the target methods; testing target methods also tests helper methods. The numbers of non-comment non-blank (ncnb) lines of source code, branches, and mutants are for all those methods.

8.3 Bounded-Exhaustive Testing

We next present the experiments that evaluate bounded-exhaustive testing of data structures. Figure 8-3 lists the methods that we test with bounded-exhaustive test suites, i.e., with test suites that contain all nonisomorphic inputs within a given scope. We use Korat to generate such inputs for the *target* methods. These methods implement the standard operations on their corresponding data structures [23]. Testing the correctness of outputs for these methods also tests some *helper* methods that either directly the target methods or their specifications invoke.

We first discuss the relative quality of generated test suites. We measure how mutation coverage and code coverage vary with the scope. We then discuss the performance of Korat for generating these test suites and the performance of checking method correctness. It may seem unusual that we discuss the quality of test suites before discussing their generation, but the quality actually determines the scope for which we should evaluate generation by Korat. We finally compare bounded-exhaustive testing with randomly selected test inputs. The experimental results support the four hypotheses listed at the beginning of this chapter.

8.3.1 Mutation

For mutation testing, we use Ferastrau (Chapter 7). We instruct Ferastrau to generate between 200 and 300 mutants for each benchmark, mutating the target methods and the helper methods that they directly invoke, but not the helper methods that only the specifications invoke. Figure 8-3 shows the number of mutants for each benchmark. Figure 8-4 shows mutation coverage for several scopes. (The numbers for smaller scopes are in Appendix A.) For all benchmarks but `FibHeap` and `TreeMap`, inputs in the largest scopes kill over 90% of the mutants.

We inspected a selection of the mutants that survive for these two benchmarks to detect if they are syntactically different but still semantically equivalent to the original program, and thus no input could kill them. Since we implemented `FibHeap`, we were able to determine that all inspected mutants are indeed semantically equivalent. Due to the complexity of `TreeMap` and our lack of familiarity with it, we were unable to definitely establish the equivalence for all inspected mutants. However, we also tested surviving mutants for inputs of scope eight and all mutants still survived, increasing our confidence that they are indeed semantically equivalent.

We conclude that these results indeed support the Mutation Hypothesis: There is a certain small mutation-adequate scope, i.e., bounded-exhaustive test suite for this scope can detect almost all nonequivalent mutants. For our set of benchmarks, this mutation-adequate scope turns out to be a number between three and seven.

8.3.2 Coverage

Figure 8-4 also shows the code coverage that bounded-exhaustive test suites achieve, and Figure 8-5 shows graphs that relate scope with the statement code coverage and mutation coverage. We measure code coverage for all target and helper methods, since testing executes all these methods. For more than half of the benchmarks, Korat generates test inputs that achieve 100% coverage, both for statements and branches. For other benchmarks, the coverage is not 100%, primarily because no input for target methods could trigger some exceptional behavior of helper methods.

For example, the target method `reverse` for lists creates a `ListIterator` and invokes helper methods on it. In general, these helper methods could throw exceptions, such as `ConcurrentModificationException` or `NoSuchElementException`, typically when used in an "incorrect" way. However, the target methods never invoke the helper methods in such a way. In terms of JML specifications, the target methods invoke the helper methods in pre-states that satisfy the precondition for `normal_behavior` and not for `exceptional_behavior`.

For all benchmarks, the minimum scope that is coverage-adequate is not mutation-adequate, i.e., the test inputs for the scope are not sufficient to kill almost all mutants.

benchmark	scope	#	time	code coverage		mutants
		inputs	[sec]	st. [%]	br. [%]	killed [%]
SearchTree	5	1880	0.63	100.00	100.00	98.52
	6	8772	0.85	100.00	100.00	99.26
	7	41300	1.65	100.00	100.00	99.26
HeapArray	5	15352	0.76	96.55	94.44	89.78
	6	118251	2.25	100.00	100.00	96.35
	7	1175620	17.95	100.00	100.00	96.71
BinHeap	5	16848	1.08	100.00	100.00	94.86
	6	159642	5.01	100.00	100.00	95.89
	7	2577984	76.36	100.00	100.00	96.91
FibHeap	3	1632	0.65	95.70	98.39	75.08
	4	34650	1.48	95.70	98.39	81.48
	5	941058	23.77	100.00	100.00	88.88
DisjSet	3	456	0.47	100.00	100.00	88.47
	4	18280	0.81	100.00	100.00	95.06
	5	1246380	20.32	100.00	100.00	95.06
LinkedList	5	2584	0.66	90.57	84.38	99.59
	6	11741	0.88	90.57	84.38	99.59
	7	58175	1.94	90.57	84.38	99.59
SortedList	5	7427	0.90	92.50	89.66	96.53
	6	73263	2.97	92.50	89.66	97.40
	7	1047608	38.31	92.50	89.66	97.40
TreeMap	5	1150	0.68	100.00	91.49	87.37
	6	3924	0.81	100.00	91.49	89.76
	7	12754	1.16	100.00	91.49	89.76
HashSet	5	3638	0.78	100.00	100.00	91.39
	6	12932	1.05	100.00	100.00	91.80
	7	54844	2.09	100.00	100.00	92.21
AVTree	3	1702	1.19	88.23	84.61	75.12
	4	27734	8.77	94.12	92.31	91.21
	5	417878	134.92	94.12	92.31	93.65

Figure 8-4: Testing target methods with bounded-exhaustive test suites. For each benchmark and scope, we tabulate the number of test inputs, the time for testing the code for all these inputs, code coverage (statement and branch), and mutation coverage.

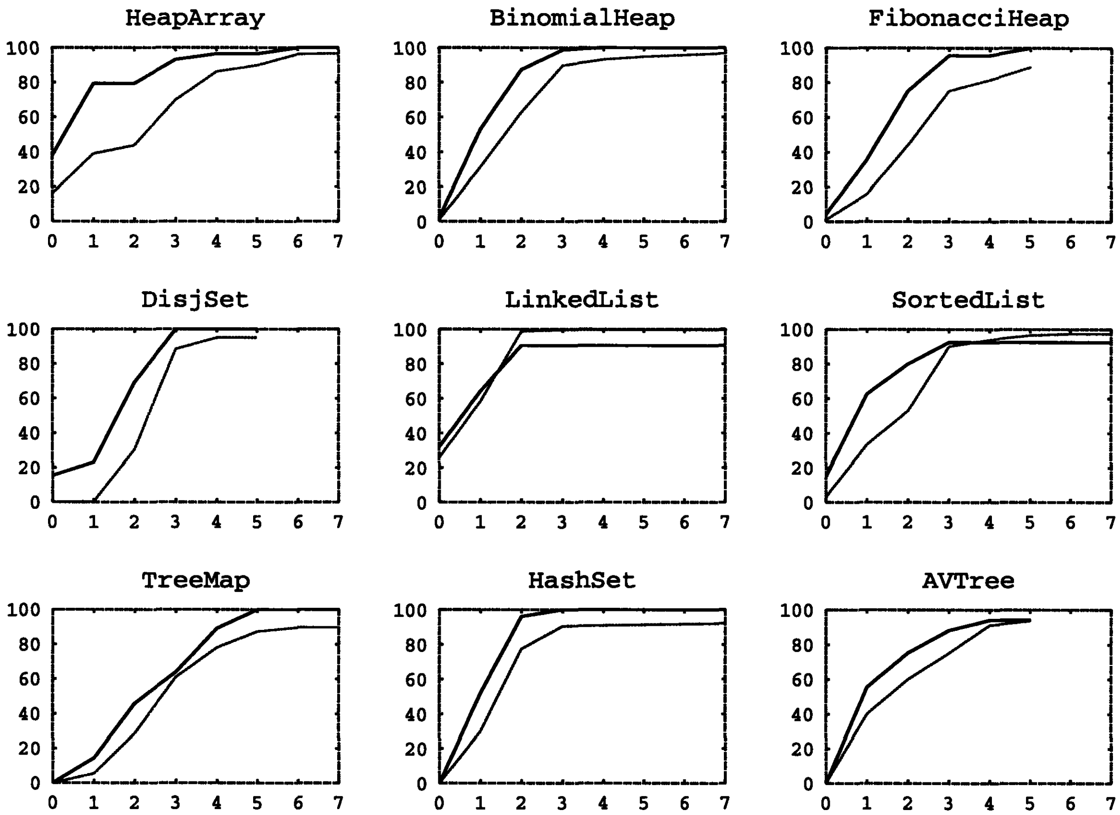

Figure 8-5: Variation of code coverage and mutating killing with scope. Thick line shows the statement code coverage and thin line the mutation coverage. For *all* benchmarks, Korat generates inputs that achieve the maximum coverage that is possible without directly generating inputs for helper methods.

We can see that increasing the scope after complete coverage increases mutation coverage. This supports the Coverage Hypothesis: The mutation criterion is stronger than the code coverage criterion, i.e., mutation-adequate scope is larger than code-coverage adequate scope. This is not surprising; it is well-known that complete statement or branch coverage (or for that matter, any coverage criteria) does not guarantee the absence of faults. Walsh [111] also showed that mutation adequacy is a stronger criterion than statement and branch adequacy. Our experiments reaffirm this, showing that mutation is stronger than code coverage for bounded-exhaustive testing of data structures.

As an illustration, consider the following code snippet from `SearchTree.remove`:

```
Node temp = left;
while (temp.right.right != null) {
    temp = temp.right;
}
```

Suppose that the mutation changes only the loop body:

```
Node temp = left;
while (temp.right.right != null) {
    temp = left.right;
}
```

For test inputs that make the loop execute zero or one times, the original program and the mutant have the same behavior. For trees with up to four nodes, the loop cannot execute more than once, and these trees achieve complete coverage for `remove`. However, for a tree with five nodes, the loop executes twice in the original program, while the mutant loops infinitely. (Recall that Ferastrau detects and kills such mutants.) Because of this, we do not consider the effectiveness of Korat in the scopes that are coverage adequate, but in the scopes that are mutation adequate.

8.3.3 Feasibility

Figure 8-6 shows the performance of Korat for generating test inputs for our data-structure benchmarks. Korat generates all nonisomorphic valid structures within the given scope, i.e., all test inputs up to the given size bound. The tabulated number of inputs is the sum of inputs for *all* target methods for each benchmark. Also, the generation times in Figure 8-6 and the testing times in Figure 8-4 are the sums of times for all target methods. We use Korat to separately generate inputs for each method. However, when two methods have the same precondition (e.g., `remove` and `add` for `SearchTree`), we could reuse the inputs and thus reduce the generation time.

For each benchmark, the largest tabulated scope is mutation adequate. In all cases, Korat completes generation in less than two minutes, often in just a few seconds. All times are elapsed real times from the start of Korat to its completion, including the JVM initialization that takes around 0.25*sec*. The use of dedicated generators reduces the generation times from a few percent up to 70% (for `SearchTree` and scope seven). Since dedicated generators have a higher overhead, their use sometimes slightly increases the generation time, especially for very small scopes.

Since JML specifications are executable, we can also measure *specification coverage* [16] as the code coverage of the predicate that corresponds to the method's precondition, for example, `removePre` from Section 6.1.2. Figure 8-6 presents coverage measured during Korat's generation of valid structures for the predicate and thus valid test inputs for the method. For most benchmarks, the tabulated scopes achieve complete specification coverage, both for statements and branches. It is not always 100%, because finitizations do not even put for fields some values that do not satisfy the predicate (e.g., `finSearchTree` from Section 3.2.2 does not put negative values for the tree size). In this setting, specification coverage becomes adequate for smaller scopes than implementation coverage, which means that specification coverage is a weaker criterion than implementation coverage.

We additionally measured the precondition/predicate coverage by executing the predicate for all valid structures, and not during Korat's generation. We do not present this coverage in Figure 8-6, because it is even weaker than specification coverage. Namely, this coverage achieves lower absolute percentage than the specification coverage—for example, some statements in the predicates, such as `return false`,

		ded. generators				
		without	with			
benchmark	scope	time	time	spec. coverage		#
		[sec]	[sec]	st. [%]	br. [%]	inputs
SearchTree	5	1.00	0.86	94.74	96.67	1880
	6	2.00	1.12	94.74	96.67	8772
	7	9.64	2.80	94.74	96.67	41300
HeapArray	5	0.80	0.78	90.00	92.86	15352
	6	1.48	1.28	90.00	92.86	118251
	7	7.78	7.00	90.00	92.86	1175620
BinHeap	5	2.04	1.88	97.67	98.00	16848
	6	4.57	3.79	97.67	98.00	159642
	7	37.06	29.30	97.67	98.00	2577984
FibHeap	3	1.70	1.71	97.78	98.28	1632
	4	2.70	2.19	97.78	98.28	34650
	5	15.48	14.29	97.78	98.28	941058
DisjSet	3	0.61	0.62	100.00	100.00	456
	4	0.88	1.02	100.00	100.00	18280
	5	11.56	10.48	100.00	100.00	1246380
LinkedList	5	1.16	1.09	100.00	100.00	2584
	6	1.23	1.23	100.00	100.00	11741
	7	1.67	1.61	100.00	100.00	58175
SortedList	5	1.76	1.70	100.00	100.00	7427
	6	3.20	3.07	100.00	100.00	73263
	7	23.90	22.54	100.00	100.00	1047608
TreeMap	5	1.06	1.00	100.00	100.00	1150
	6	1.62	1.40	100.00	100.00	3924
	7	3.96	2.42	100.00	100.00	12754
HashSet	5	1.06	1.06	89.47	92.31	3638
	6	1.60	1.40	89.47	92.31	12932
	7	4.06	3.58	89.47	92.31	54844
AVTree	3	0.53	0.49	96.67	96.88	1702
	4	3.49	2.19	96.67	96.88	27734
	5	87.46	43.74	96.67	96.88	417878

Figure 8-6: Performance of Korat for generating test inputs. For each benchmark and scope, we tabulate the times to generate bounded-exhaustive test suites without and with dedicated generators, specification coverage (statement and branch), and the number of test inputs. All times are elapsed real times in seconds from the start of Korat to its completion.

		random	bounded-exhaustive	
benchmark	scope	mutants killed [%]	scope-1	scope
`SearchTree`	7	99.26	=	=
`HeapArray`	7	95.99	<	<
`BinHeap`	7	95.10	<	<
`FibHeap`	5	86.87	>	<
`DisjSet`	5	95.06	=	=
`LinkedList`	7	99.59	=	=
`SortedList`	7	96.40	<	<
`TreeMap`	7	89.08	<	<
`HashSet`	7	91.39	<	<
`AVTree`	5	93.17	>	<

Figure 8-7: Comparison of bounded-exhaustive testing and random testing. Comparison of mutation coverage for bounded-exhaustive test suites for scopes s and $s - 1$ with randomly chosen test suites from scope s with the same number of test inputs as test suites in scope $s - 1$; '=' means that both suites are equally good, '<' that random is worse, and '>' that random is better.

are never executed for valid structures—but becomes adequate (and achieves the maximum possible percentage) for even smaller scopes than the specification coverage.

We next discuss the testing times presented in Figure 8-4. The postconditions for all methods specify typical partial correctness properties; they require the resulting data structures (1) to be valid and (2) depending whether the method adds or removes an element, to contain or not contain the input element. The testing times depend on the complexity of methods and their postconditions, so they are not under the control of Korat. Overall, the testing times are within the same magnitude as the corresponding generation times across all benchmarks.

These results support the Feasibility Hypothesis: Korat can effectively generate bounded-exhaustive test suites for the mutation-adequate scope, and it is practical to check correctness for these test suites, despite the potentially large number of test inputs. Korat is able to efficiently generate the test inputs due to the pruning based on accessed fields in the precondition predicates and the elimination of isomorphic test inputs. Testing only nonisomorphic inputs is very important in reducing the testing time. In conclusion, Korat can be effectively used for bounded-exhaustive testing of our benchmarks.

8.3.4 Randomness

Bounded-exhaustive testing uses test suites with all nonisomorphic valid test inputs within a given scope. A question that naturally arises is whether some test-selection strategy could reduce the size of bounded-exhaustive test suites without sacrificing their quality. We next evaluate the simplest test-selection strategy, namely, uniform random sampling.

Consider one benchmark, and let $T(s)$ be the set of all nonisomorphic valid test inputs within scope s. From $T(s)$, we randomly choose a subset $R(s)$ whose cardinality is the same as the cardinality of $T(s-1)$. We then compare the quality of $R(s)$ against $T(s-1)$ and $T(s)$. For comparison, we use mutation coverage, because this criterion is stronger than code coverage. To offset the influence of random selection, we average mutation coverage over five random samples.

It is important to point out that randomly chosen test inputs are also generated by Korat. For predicates that express constraints on structurally complex inputs, it is not feasible to simply generate random inputs that satisfy the predicate, because the input spaces are very sparse (Section 8.2).

Figure 8-7 shows the comparison of mutation coverage for randomly sampled and bounded-exhaustive test suites. The randomly sampled test suites achieve lower mutation coverage for half of the benchmarks, the same mutation coverage for three of the benchmarks, and higher mutation coverage for only two benchmarks, `FibHeap` and `AVTree`. This supports the Randomness Hypothesis: Bounded-exhaustive test suites for a mutation-adequate scope can be more effective than random test suites with the same number of inputs randomly selected from a larger scope. There may be, however, a test-selection strategy different from random that can reduce the size of bounded-exhaustive test suites without reducing their quality. This strategy would need to take into account the specifics of the code under test and its specification.

Chapter 9

Comparison with Testing Based on Declarative Predicates

This chapter compares testing based on imperative predicates with testing based on declarative predicates as embodied in the TestEra tool [55, 73]. We present this comparison is detail because we were involved in the development of TestEra, and our work on TestEra and Korat mutually enhanced each other. Chapter 10 discusses other related work.

Section 9.1 gives an overview of TestEra. TestEra uses the Alloy language [52] to express constraints on test inputs and Alloy Analyzer [51] to solve these constraints and generate test inputs. This section uses an example to illustrate Alloy and discusses qualitative differences between imperative and declarative predicates, namely, Java and Alloy predicates. Section 9.2 presents a quantitative comparison of Korat and TestEra. Section 9.3 summarizes the comparison.

9.1 Overview of TestEra

TestEra is a tool for bounded-exhaustive testing of programs. The user provides TestEra with a predicate that characterizes test inputs of interest and a bound for the input size, and TestEra generates all nonisomorphic valid test inputs within the bound. If the user provides an oracle, TestEra can additionally run the code for each input and check the correctness of the output.

Unlike Korat that operates on imperative predicates, TestEra operates on declarative predicates. Specifically, TestEra uses the Alloy language [52] for predicates. Alloy is a modeling language based on the first-order logic formulas with transitive closure. Alloy is suitable for modeling structural properties of software. We have developed Alloy models of several data structures [57, 73], corresponding to the `repOk` predicates for class invariants in Korat.

TestEra uses Alloy Analyzer to generate test inputs from Alloy predicates. Each Alloy predicate expresses constraints on several relations. Alloy Analyzer [51] is an automatic tool for analyzing Alloy predicates. Given a predicate and a scope—a bound on the number of atoms in the universe of discourse—Alloy Analyzer can

```
boolean repOk() {
  // checks that the object graph is a tree
  all n: root.*(left + right) {
    n !in n.^(left + right) // no directed cycle
    sole n.~(left + right)  // at most one parent
    no n.left & n.right     // distinct children
  }
  // checks that size is consistent
  size = #root.*(left + right)
  // checks that data is ordered
  all n: root.*(left+right) {
    all nl: n.left.*(left + right) | LT(nl.info, n.info)
    all nr: n.right.*(left + right) | GT(nr.info, n.info)
  }
}
```

Figure 9-1: Example declarative predicate in Alloy. This predicate expresses the class invariant for the SearchTree class.

generate all mostly nonisomorphic solutions to the predicate. Each solution gives values to the relations in the predicate to make it evaluate to true. Alloy Analyzer operates in three steps: (1) it translates Alloy predicates into propositional formulas, i.e., constraints where all variables are boolean; (2) it uses off-the-shelf SAT solvers to find solutions for the propositional formulas; (3) it translates each solution from the propositional domain into the relational domain. TestEra can additionally translate each Alloy solution into an actual test input [55].

We developed the first version of TestEra [56] before Korat, and Alloy Analyzer has influenced the design of Korat, in particular finitizations. Setting the scope in Alloy Analyzer corresponds to setting the finitization parameters in Korat. Additionally, Alloy models of data structures declare field types; these declarations correspond to setting field domains in Korat finitizations. Our work on Korat also influenced generation of nonisomorphic solutions in Alloy Analyzer (Section 9.2.1).

9.2 Ease of Writing Predicates

We next present an example that illustrates Alloy predicates. Through the example, we discuss the relative ease of writing Alloy and Java predicates for the same constraints that describe structural properties. While writing predicates in either Alloy or Java is always easier than, say, writing them directly as propositional formulas, we cannot currently claim that writing predicates in either Alloy or Java is always easier than writing them in the other language.

Recall the SearchTree example from Section 2.1. Appendix A shows the full Java code for the repOk method that checks the class invariant for SearchTree. This invariant has three properties: (1) the underlying graph is indeed a tree, (2) the value of the size is the same as the number of nodes, and (3) the data is ordered for binary search. Figure 9-1 shows an Alloy predicate, adapted from Khurshid [55], that specifies these properties. The predicate specifies the first and the third properties with universally quantified formulas and the second property with an equality. Alloy implicitly conjoins the formulas. We explain details of the first formula.

The Alloy expression `root.*(left + right)` denotes the set of all nodes reachable from the `root` along the fields `left` and `right`. Alloy represents each Java field with a relation, for example, `left` is a binary relation from `Node` objects to `Node` objects. The empty set represents `null`; if a field f of an object o has value `null`, there is no pair for o in the relation for f. The Alloy operations '`+`', '`*`', and '`.`' denote union, reflexive transitive closure, and relation composition, respectively. The first formula specifies that for every node reachable from `root`, three subformulas hold. First, node is not reachable from itself; '`^`' is transitive closure, '`in`' is membership, and '`!`' is negation. Second, node does not have two incoming pointers; '`~`' is relation transposition, and '`sole`' checks that the set has at most one element. Third, node points to different children; '`&`' is set intersection, and '`no`' checks that the set is empty.

We can see that Alloy predicates are much more succinct than Java predicates; to specify the invariant in this example takes about three times less lines in Alloy than in Java. Being declarative, Alloy allows the operations that are not easy to compute with an imperative predicate, e.g., '`~`' finds the inverse of all pointers. However, Alloy has a different style for writing predicates than Java, and Alloy does not have built-in support for several Java constructs [55], e.g., primitive values or exceptions. We have found that these issues make it hard for programmers familiar with Java to write Alloy predicates. On the other hand, our experience also shows that for researchers familiar with Alloy, it is often easier to write Alloy predicates than Java predicates.

9.2.1 Nonisomorphic Inputs

We discuss generation of nonisomorphic inputs in TestEra and compare it with Korat. Alloy Analyzer can automatically add so called *symmetry-breaking predicates* [3, 24] to the propositional formula such that different solutions to the formula represent (mostly) nonisomorphic solutions of the Alloy predicate. However, there is a tradeoff in adding symmetry-breaking predicates: if too few are added, the SAT solver spends more time generating isomorphic solutions, but if too many are added, the formula becomes too big, and the solver spends more time exploring it. By default, Alloy Analyzer adds symmetry-breaking predicates that eliminate *most* isomorphic solutions and significantly decrease the solving time [96] compared to the time without symmetry-breaking predicates. If Alloy Analyzer automatically added symmetry-breaking predicates to eliminate *all* isomorphic solutions, the solving time would significantly increase compared to the time with default symmetry-breaking predicates. Inspired by Korat, we developed a methodology for manually adding symmetry-breaking predicates to Alloy to eliminate all isomorphic solutions and still reduce the solving time [57].

The methodology requires the user to define a total order on all objects that can appear in the inputs and a traversal that puts the order on the objects that are actually reachable from the root. The combination of these two orders induces

```
boolean breakSymmetries() {
  // if tree is nonempty, root is the first node in linear order
  root in OrdFirst(Node)
  // define traversal and require nodes to be visited in order
  all n: root.*(left + right) {
    some n.left => n.left = OrdNext(n)
    some n.right && no n.left => n.right = OrdNext(n)
    some n.right && some n.left =>
      n.right in OrdNext(n.left.*(left + right))
  }
}
```

Figure 9-2: Manual symmetry-breaking predicate in Alloy. The user can write this example predicate to eliminate isomorphic solutions for `SearchTree` invariant.

a lexicographic order, similar to that used in Korat to compare input structures (Section 3.2.5). Conjoining the original predicate, which puts constraints on the input, and the symmetry-breaking predicate produces a new predicate whose solutions are exactly nonisomorphic inputs of the original predicate.

Figure 9-2 shows an example symmetry-breaking predicate taken from Khurshid [55]. Alloy provides a library of functions for imposing order. The function `OrdFirst` takes a set, imposes a total order on the elements of the set, and returns the smallest element from the set. The function `OrdNext` takes a set, which can be a singleton, and returns the smallest element that is greater than all the elements in the set. The predicate uses these functions to impose a pre-order [23] for the nodes in the tree: all children of a node `n` are larger than `n` itself, and all children in the left subtree are smaller than the children in the right subtree. Note that this ordering is with respect to the node identities, not the values in the nodes; the `repOk` predicate actually imposes an in-order [23] on the values.

Symmetry Breaking

We next compare symmetry breaking in Korat and TestEra. We first discuss the advantages and disadvantages of automatic symmetry breaking in these two tools. We then present an example that uses manual symmetry breaking for generating directed acyclic graphs (DAGs).

Korat automatically breaks all symmetries if inputs are structures, i.e., rooted and deterministically edge-labeled (object) graphs (Definition 2). However, Korat does not automatically break all symmetries if inputs are general graphs, i.e., a node/object can have several outgoing edges/fields with the same label. At the concrete level, object graphs in Java do not have such nodes. But there can be such nodes if inputs are not rooted (in which case, we can introduce a special root node and connect it to every other node with edges that share the same label) or more broadly, when we view sets at the abstract level (such that one objects can have fields that are sets of objects). This is the case, for example, for (general) DAGs: every node has a set of successor nodes. More importantly, such general graphs could represent inputs that arise in practice in system testing, even though structures could suffice to represent inputs that arise in unit testing.

In theory, TestEra can automatically break *all* symmetries for structures and even for general graphs. For general graphs with n nodes, therefore, TestEra could automatically generate up to $n!$ less graphs than Korat, and thus TestEra would be more practical than Korat when inputs are general graphs. But breaking all symmetries can significantly slow down TestEra's generation (and would correspond to generating all inputs with Korat and then filtering out isomorphic ones). In practice, TestEra automatically breaks *some* symmetries for structures and general graphs. Experimental results for several structures [55] show that the number of solutions that TestEra with default symmetry breaking generates is from 1.5 times to 177 times larger than the number of nonisomorphic solutions. The user of TestEra can manually provide symmetry-breaking predicates to efficiently break all symmetries. When structures have a tree backbone [55,80], the user can easily follow the above methodology. However, it is not clear how to break all symmetries for graphs with no tree backbone such as DAGs.

Sullivan et al. [104] recently presented a case study that used bounded-exhaustive testing to detect faults in a real application. Their study used TestEra to generate test inputs from Alloy predicates. Without considering the details of the study, we mention that one of the predicates can be viewed as specifying the invariant for rooted DAGs, namely, that there are no directed cycles among the nodes reachable from the root. An Alloy expert developed the predicate for this invariant and also a symmetry-breaking predicate that eliminates most isomorphic DAGs. We developed a corresponding Java predicate for this invariant and also manually added symmetry breaking that eliminates some isomorphic DAGs. Our use of Korat revealed that the symmetry-breaking predicate written in Alloy was incorrect: it eliminated not only nonisomorphic inputs but also some valid inputs.

In summary, automatic symmetry breaking is preferable; manual symmetry breaking puts additional burden on the user and is error-prone. Korat can automatically break *all* symmetries for structures, while TestEra can automatically break *some* symmetries even for general graphs. It would be worthwhile to explore how the techniques from these tools could enhance each other: how to automatically break (1) some symmetries for general graphs in Korat and (2) all symmetries for structures in TestEra (without slowing down the generation).

9.3 Efficiency of Generation and Checking

We next compare the performance of testing based on Korat and TestEra. More precisely, we present the experimental results for performance of Korat and TestEra in generating data structures and briefly discuss correctness checking. We compared Korat and TestEra earlier [13], but both tools have evolved since then, and this comparison includes the latest results.

Figure 9-3 summarizes the performance comparison. We took the timing results for TestEra from Khurshid [55] and performed all Korat experiments on the same

benchmark	size	solutions	Korat	SAT
			[sec]	[sec]
SearchTree	7	429	5.69	6.46
	8	1430	48.54	40.46
	9	4862	377.83	548.69
	10	16796	3480.63	—
HeapArray	6	13139	0.72	5.10
	7	117562	2.03	62.62
	8	1005075	15.30	1171.64
	9	10391382	151.50	—
LinkedList	7	4140	0.81	4.76
	8	21147	0.94	36.52
	9	115975	2.39	304.97
	10	678570	11.62	—
TreeMap	7	35	2.35	110.42
	8	64	8.78	254.13
	9	122	44.25	741.55
	10	260	242.81	—
HashSet	7	1716	2.33	31.52
	8	6435	8.85	151.42
	9	24310	38.99	511.51
	10	92378	177.47	—

Figure 9-3: Comparison of Korat and SAT. For each benchmark and size, we tabulate the number of satisfying solutions and the times that Korat and SAT take to generate these solutions.

Linux machine with a 1.8 GHz Pentium 4 processor using Sun's Java 2 SDK 1.3.1 JVM. For each benchmark for which the TestEra results are available [55], we tabulate for several sizes the number of data structures generated and the times that Korat and SAT solver take to generate these structures. TestEra uses Alloy Analyzer, which in turn uses off-the-shelf SAT solvers for generation. The results in Figure 9-3 are for the mChaff SAT solver [82].

All times in Figure 9-3 are the total elapsed real times (in seconds) that each experiment took from the beginning to the end. For Korat, we present times without dedicated generators, i.e., Korat can generate data structures even faster with dedicate generators. Korat times include the JVM start-up time, which is about 0.25*sec*. For SAT, the times also include the start-up time, but it is smaller than for Korat because the experiments with SAT use a precompiled binary for mChaff. However, SAT solving times do not show the total time that Alloy Analyzer takes: before invoking the SAT solver, Alloy Analyzer needs to translate the Alloy predicate into a propositional formula, which can take from 0.25*sec* up to over a minute, depending on the scope.

In all cases but SearchTree for size eight, Korat outperforms SAT. Korat is not only faster for smaller sizes, but it also completes generation for larger sizes; SAT did not complete generation after an hour for the largest listed size for each benchmark. Since Java and Alloy predicates encode the same solutions, there are two reasons that could explain why Korat is faster than SAT. First, it may be that the current (implementation of the) translation of Alloy into propositional formulas generates unnecessarily large formulas. Second, it may be that SAT solvers are very inefficient for solution enumeration, i.e., for generating all solutions of a given formula.

SAT solvers are not optimized for solution enumeration; actually only a few solvers even support solution enumeration [57]. Instead, SAT solvers are optimized to find one solution or show that none exists, because this is what suffices in most applications of SAT solvers. We advocated the importance of efficient solution enumeration for testing [57], and our results also showed that mChaff was the fastest for solution enumeration, faster than the relsat SAT solver [7] and also faster than using Boolean Decision Diagrams, in particular the CUDD package [99], instead of SAT solving.

Figure 9-3 presents the results for solving propositional formulas from Alloy predicates with manually added symmetry-breaking predicates. Our previous work [13] compared Korat with propositional formulas with automatically added symmetry-breaking predicates. The results showed that SAT solving is then even more slower than Korat. One reason is that SAT can generate a much greater number of solutions than Korat, which takes a greater amount of time by itself. One way to reduce the number of solutions is to automatically add more symmetry-breaking predicates, but this would further increase the size of the propositional formulas, and it is not clear how this trade-off would affect SAT solving.

We finally discuss the performance of checking code for bounded-exhaustive suites with Korat and TestEra. Checking consists of three steps: (1) preparing the actual test inputs, (2) running the code, and (3) checking the output. For step (1), Korat and TestEra take about the same time: Korat generates an actual test input, i.e., a Java object graph, directly from each valid candidate (Section 3.2.4), and TestEra creates Java objects and set their fields according to each boolean solution [55]. For step (2), Korat and TestEra take the same time for the same inputs. For step (3), Korat checks the correctness directly evaluating Java assertions at run-time, whereas TestEra translates Java object graphs into the Alloy domain to evaluate oracles written in Alloy [55]. For semantically equivalent oracles, TestEra typically has a higher overhead, due to the translation, and thus Korat has faster checking.

9.4 Summary

So far, there is no clear winner: we view Korat and TestEra as complementary approaches. Our main argument for developing Korat, after we already developed TestEra, was simple: for Java programmers who are not familiar with Alloy, it is easier to write a repOk predicate in Java than in Alloy. However, to really compare which language is easier to use in general, we would need to conduct an extensive user study [79]. It is out of the scope of this dissertation, but we list some questions that the study should answer:

- (Development) How much time does it take to write a predicate from scratch, say from a given description in English? How much time does it take to get the predicate right, as opposed to writing something that is almost right?

- (Maintenance) How much time does it take to modify a given predicate if the description somewhat changes? How much time does it take for one person to understand/modify a predicate written by another person?

- (Reuse) How much reuse is there from one problem to another and how good libraries can we build?

- (Efficiency) How well can Korat/TestEra generate test inputs from predicates? How easy is it for the user to optimize that?

Regarding the efficiency of test-input generation, our results show that well written imperative predicates lead to faster generation of structures in Korat than in TestEra. Before conducting these experiments, we expected that Korat would generate test inputs slower than SAT and the whole TestEra. Our intuition was that Korat depends on the executions of `repOk` to "learn" the invariant of the structures, whereas SAT can "inspect" the entire formula, which represents the invariant, to decide how to best search for a solution. The experimental results show that our assumption was incorrect—Korat generates structures much faster than SAT. However, the generation time for Korat is sensitive to the way that predicates are written (Section 11.1.2, and it is unclear without a user study whether Korat or TestEra would be more efficient for typical predicates. Moreover, it may be possible to further optimize the translation of Alloy into propositional formulas and solution enumeration in SAT to outperform Korat. Finally, when inputs are not structures (Section 9.2.1), TestEra automatically break symmetries that Korat does not, and thus TestEra generates inputs faster.

Chapter 10

Related Work

This chapter reviews the work related to Korat and bounded-exhaustive testing. Section 10.1 discusses other projects on automating generation of test inputs. Section 10.2 compares bounded-exhaustive testing with static program analysis, in particular shape analysis that considers data-structure benchmarks similar to those used in Korat experiments. Section 10.3 compares bounded-exhaustive testing with software model checking.

10.1 Automated Test-Input Generation

There is a large body of research on automating generation of test inputs. The proposed approaches generate inputs from the code under test or from other artifacts such as specifications or models of the system under test and its inputs. An early paper by Goodenough and Gerhart [37] emphasizes the importance of specification-based testing, in particular in detecting errors of omission. Many projects propose techniques and tools that automate test-input generation from specifications, such as Z specifications [30, 47, 103], UML statecharts [84], or ADL specifications [16, 94]. Unlike Korat, these tools do not consider linked data structures with complex invariants.

Researchers have also been exploring the use of constraint solving for test-input generation since the 1970s [29,39,49,50,61,63,68]. However, those approaches focused on constraints on primitive data, typically integers and arrays of integers. One reason for this is that for some types of constraints on integers, for example linear arithmetic, there are decision procedures that can check the satisfiability of constraints without requiring a bound on the size. We instead consider constraints on more complex, structural data, but require the user to provide a bound on the size.

To the best of our knowledge, TestEra was the first tool for bounded-exhaustive generation of structurally complex test inputs from (declarative) constraints. Chapter 9 compares Korat and TestEra [55, 73] in detail. TestEra provided the initial motivation for the work on Korat. In turn, TestEra was inspired by the work on the Alloy language [52] and the bounded-exhaustive, SAT-based checking done by Alloy Analyzer [51].

There are many tools that generate test inputs from different test descriptions, for example from generators, grammars, or models (such as finite state machines). QuickCheck [20] is a tool for testing Haskell programs. It requires the tester to write Haskell functions that can produce valid test inputs, similar to the examples presented in Chapter 5. Executions of such functions with different random seeds produce different test inputs. Korat differs in that it requires only an imperative predicate that characterizes valid test inputs and then uses a general-purpose search to generate *all* valid inputs. DGL [75] and lava [97] generate test inputs from context grammars. They were used mostly for random testing, although they could in principle also exhaustively generate test inputs, but that would boil down to writing special-purpose generators.

The AsmL Test Tool (AsmLT) [33] contains a solver for imperative predicates inspired by Korat. AsmLT generates test inputs from AsmL specifications [42], and the solver solves predicates written in the AsmL language. The first version of AsmLT [40] used only finite-state machines to generate test inputs as sequences of (method) calls. The solver additionally enables AsmLT to generate data structures. The use of AsmLT at Microsoft resulted in some of the largest case studies on bounded-exhaustive testing so far.

A popular approach to generate test inputs as sequences of method calls is to use (smart) random generation. Tools for Java such as Jtest [89] (a commercial tool) or JCrasher [25] and Eclat [88] (two research prototypes) implement this approach. The advantage of random generation is that, given the same amount of time, it produces (some) larger inputs than bounded-exhaustive generation; however, the disadvantage is that these inputs are not exhaustive. Our experiments show that random testing with the same number of inputs from a larger scope is often worse than bounded-exhaustive testing with all inputs from a smaller scope (Section 8.3.4), but more evaluation is necessary to compare random testing and bounded-exhaustive testing. Another advantage of random generation is that it does not require imperative predicates such as `repOk` methods that check data-structure invariants. However, random generation of test inputs as sequences requires that methods be composed into sequences. Such methods are available for abstract data types, but not, for example, for XML documents that arise in XPath queries.

Cheon and Leavens [17] propose JMLunit, a tool that automatically translates JML specifications into test oracles for JUnit [9], a popular framework for unit testing of Java classes. JUnit automates test execution and error reporting, but requires programmers to provide test inputs and test oracles. JMLunit can also generate test inputs; it requires the programmer to provide possibilities for each method argument and uses the entire Cartesian product to generate test inputs, which cannot handle very large input spaces. Additionally, JMLunit does not generate complex inputs, but requires the programmer to create and provide them. Our tool based on Korat and JMLunit further automates and optimizes generation of test inputs, thus automating the entire testing process.

JMLAutoTest [117] is a tool that builds on Korat and JMLunit. JMLAutoTest additionally allows the user to provide methods for comparison of equivalent inputs, such as `equals`, instead of using built-in isomorphism. It is straightforward to generate only nonequivalent inputs if we first generate all inputs and then do a pairwise comparison *after* the generation. It is an open problem how to avoid equivalent inputs during the generation.

AETG [21] is a popular system for generating test inputs that cover all pair-wise (or n-wise) combinations of test parameters that correspond to object fields in Korat. Using pair-wise testing is applicable when parameters are relatively independent. However, fields of objects in object graphs are dependent. We can view Korat as an efficient approach to generate all inputs when n is the same as the number of parameters. Additionally, Korat takes into account isomorphism and generates only one input from each isomorphism partition.

10.2 Static Analysis

Several projects have developed static analyses for verifying program properties. We consider some of the projects for checking properties of complex data structures. The Extended Static Checker (ESC) [32] uses a theorem prover to verify partial correctness of Java classes annotated with JML specifications. ESC can verify absence of errors such as null pointer dereferences, array bounds violations, and division by zero. However, tools like ESC do not verify complex properties of linked data structures.

The Three-Valued-Logic Analyzer (TVLA) [93] was the first system to implement a shape analysis, which verifies that programs with destructive updates preserve structural invariants of linked data structures. TVLA was initially used to analyze programs that manipulate singly and doubly linked lists, circular lists, as well as some sorting programs. Recent work [118] extends TVLA to handle more properties of the content of data structures and uses symbolic computation to automatically derive transfer functions, thus reducing the amount of work that the user needs to do.

The Pointer Assertion Logic Engine (PALE) [80] can verify a large class of data structures that have a spanning tree backbone, with possibly additional pointers that do not add extra information. These data structures include doubly linked lists, trees with parent pointers, and threaded trees. However, PALE focuses only on these structural properties and cannot verify properties that involve the elements of the structures.

Kuncak et al. proposed Role Analysis [64], a compositional interprocedural analysis that verifies similar properties as TVLA and PALE. The same authors recently developed Hob [65], an analysis framework that combines different static-analysis and theorem-proving techniques [119] to show complex data structure consistency properties. Hob was applied to programs with linked data structures such as search trees, singly and doubly-linked lists with and without an iterator, and array-based data structures such as priority queues. Among the example programs that Hob analyzed are a working Minesweeper game and the Water simulation benchmark.

Static analysis of program properties is a promising approach for ensuring program correctness in the long run. However, the current static-analysis techniques can only verify limited program properties. For example, none of the above techniques can currently verify correctness of implementations of red-black trees. Recent work on quantitative shape analysis [91] developed a specialized static analysis for verification of AVL trees. It is likely that a specialized static analysis can be also developed for red-black trees, but such an approach would require an involved development of a static analysis for each new property. Bounded-exhaustive testing, on the other hand, is very general and can verify any decidable program property, but for inputs bounded by a given size.

Jalloy [53, 108] analyzes methods that manipulate linked data structures by first building an Alloy model of Java code and then checking it in a bounded-exhaustive manner with Alloy Analyzer [51]. This approach provides static analysis, but unsound with respect to both the size of the input and the length of computation. Our approach based on Korat checks the entire computation and handles larger inputs and more complex data structures than those in [53, 108]. Further, Korat does not require Alloy, but JML specifications, and Korat does not require specifications for all (helper) methods. The advantage of Jalloy is that it can better handle some scenarios where the number of valid inputs is huge, and bounded-exhaustive testing is not applicable. Consider testing whether an identity method really implements identity, i.e., returns the input unchanged, for all inputs within some bound. Using Korat, we would need to generate all those inputs, whereas Jalloy only needs to check the formula `x == x`, which it can do efficiently, without enumerating all `x`.

10.3 Software Model Checking

There has been a lot of recent interest in applying model checking to software. VeriSoft [36] was the first system for model checking of programs written in an actual implementation language, specifically in C. While VeriSoft performs a systematic exploration of the state space without storing the visited states (stateless), CMC [83] performs a statefull exploration of C programs. Java PathFinder [109] and Bogor [90] are two systems that perform a statefull exploration of Java programs. Other projects, such as Bandera [22] and JCAT [27], translate Java programs into the input language of existing model checkers such as SPIN [46] and SMV [77]. They handle a significant portion of Java, including dynamic allocation, object references, exceptions, inheritance, and threads. They also provide automated support for reducing program's state space through program slicing and data abstraction.

However, most of the work on applying model checking to software has focused on checking event sequences and not linked data structures. Where data structures have been considered, the purpose has been to reduce the state space to be explored and not to check the data structures themselves. Korat and bounded-exhaustive testing, on the other hand, focus on checking code that manipulates linked data structures.

Recently, Khurshid et al. [58, 110] combined the Java PathFinder model checker with a Korat-like search for object references and symbolic execution for primitive values. The resulting system can be used for bounded-exhaustive testing of data structures. The initial results are promising, but the system still requires manual instrumentation for symbolic execution and has been evaluated on only two benchmarks, so it is unclear how the results generalize.

Symbolic execution is also the foundation of model checkers that automatically abstract programs into smaller models that can be effectively checked. Two such systems are SLAM [6] and BLAST [45], which abstract C programs into boolean programs. Similar systems typically do not generate test inputs, but automatically verify simple properties of programs. However, SLAM [4] and BLAST [11] were recently adapted for test-input generation. Neither of them deals with complex data structures, which are the focus of this dissertation.

Chapter 11

Conclusions

This chapter presents the concluding remarks. This dissertation has described the problem of solving imperative predicates that characterize structurally complex test inputs, presented a technique for solving imperative predicates and a tool called Korat that implements the technique, and discussed bounded-exhaustive testing of data structures. Section 11.1 presents a discussion of the overall approach. Section 11.2 gives some avenues for future work. Section 11.3 provides a summary.

11.1 Discussion

We discuss our proposed approach for solving imperative predicates and bounded-exhaustive testing, and we point out some limitations and problems with the proposed approach. We first discuss the use of imperative predicates as a language for expressing properties of structurally complex data. Although using the actual programming language has several benefits, it may be that other languages are even better for expressing properties. We then discuss the performance of Korat for generation of valid structures. Although Korat is effective for generation of structures for properties of our benchmark data structures, Korat is not effective for generation of structures for all properties. Also, it may be possible that there is a technique that can solve imperative predicates much faster than the techniques that Korat uses. We finally discuss bounded-exhaustive testing. Although it is practical to test our set of benchmarks with bounded-exhaustive test suites, this approach is not practical for all types of software.

11.1.1 Language for Predicates

We believe that building techniques and tools on concepts familiar to the intended users, such as implementation languages to programmers, is a key for immediate adoption and potential impact. For this reason, we developed a technique for solving imperative predicates written in the actual programming languages. Our anecdotal experience shows that several users found it easier to write imperative predicates for Korat than declarative predicates for TestEra (Chapter 9). Moreover, the use of the

AsmL Test Tool (AsmLT) at Microsoft also points out the importance of building on a familiar language. AsmLT first included a solver for predicates written in the AsmL language [40, 42], but the need to learn a new language was an obstacle for a wider adoption of the tool. As a result, AsmLT now includes a solver for C# predicates. (As of this writing, the new tool is available only internally.)

However, this does not mean that the existing and widely-used languages are necessarily the best for expressing properties of test inputs. It may be the case that other languages make it easier to write the properties or enable faster generation. For example, Alloy allows the succinct expression of structural properties and AsmL offers several features not directly available in C#, including parallelism, nondeterministic choice, and backtracking. The users may be better off in a long run by learning a new language, despite a potentially steep learning curve. We believe that it is worthwhile to further investigate the language design, and we present some potential directions in the next section.

11.1.2 Efficiency of Generation

The results in Chapter 8 show that it is practical to use Korat to generate structures for our set of data-structure benchmarks. It is important to consider if and how the generation could be even faster and also for what kind of properties Korat can effectively generate structures. We argue here that the generation can be faster: since Korat executes the predicate for some invalid candidate structures, it is possible to improve the performance simply by reducing the number of invalid candidates. The results with dedicated generators already support this. We discuss in the next section how we could make the generation faster.

Performance of Korat in solving a given imperative predicate depends on (1) the property that the predicate checks and (2) the way that the predicate checks the property. Performance of Korat thus depends not only on the predicate's denotational meaning—the set of valid structures for the property—but also on the actual operational behavior of the predicate. Korat prunes the search for valid structures based on the order in which the predicate accesses field of the candidate structures. For certain properties, we can (re)write the predicate such that Korat's pruning is effective, but this is not possible for all properties. We next present guidelines for writing effective predicates and then discuss properties for which this is not possible.

Guidelines for Writing Effective Imperative Predicates

We have found that Korat works well in practice for naturally written imperative predicates that efficiently check the properties: if a predicate is more efficient in checking, Korat is usually more efficient in generating valid structures for the predicate. Many practical predicates check a property that is a conjunction of several (sub)properties. We present five guidelines for (re)writing predicates and the order in which they check properties to enable Korat to more efficiently prune the search.

Immediate Termination: The basic guideline is that a predicate returns true or false as soon as it finds the input structure to be valid or invalid, respectively, without reading any additional fields. A predicate typically has to read the whole structure to determine that it is valid. This is not always the case; for example, recall from Section 6.1.2 the structures that represent inputs to the remove method and the removePre predicate for these structures. This predicate should not read the SearchTree_remove.info field, because it can determine whether a structure is valid without reading this field. A predicate can often determine that a structure is invalid based only on a part of the structure. In such situations, the predicate should immediately return false.

Read to Check: A predicate should read a field for the first time only if it is checking a property that depends on that field. The previous guideline requires that a predicate not read any field *after* the predicate determines the result, and this guideline requires that a predicate not read a field *before* it needs the field's value to determine the result. For example, repOk for SearchTree could access the size field before any other field. (It cannot, for example, access any left or right field before accessing root, because nodes are only reachable from the root.) However, repOk should not read this field before comparing its value with the number of the nodes in the tree.

Fewer Dependencies First: A predicate should check the properties that depend on fewer fields of the structure before checking the properties that depend on more fields. For example, the isTree property in our example repOk for SearchTree (Section 2.1) depends on the fields root, left, and right; and the isOrdered property depends on these fields and additionally on info. Thus, repOk checks isTree before isOrdered. For the specific predicates from Appendix A, it is necessary to check isTree before isOrdered, because isTree is a precondition for isOrdered[1]. A related guideline is that a predicate should check first the properties that other properties conceptually depend on, in this example, isTree before isOrdered, even if there are no explicit preconditions.

Local Before Global: A predicate should check local properties, which depend on a smaller part of the structure, before checking global properties, which depend on a larger part of the structure. For example, each red-black tree has to satisfy two properties for the node colors: (1) a red node should have black children and (2) the number of black nodes should be the same on every path from the root to a leaf. A predicate that checks red-black properties should check the property (1) before checking the property (2). Note that both of these properties depend on the same fields (node colors and links between nodes).

Merging Properties: A predicate can merge several properties to check them together. For example, repOk for SearchTree can check at once that the graph is a tree (treeness) and that the values are ordered (orderness). This would reduce the number of candidates that Korat explores, because it can prune the search as soon as the predicates finds either treeness or orderness to be false. Although merging properties can make generation faster, we recommend a careful and sparing use of this

[1] In general, a predicate that depends on fewer fields does not need to be a precondition for a predicate that depends on more fields; we could have written isOrdered to keep track of the visited nodes (to avoid cycles) and not depend on isTree.

guideline because it makes the predicates harder to reuse and change! For example, we cannot use a predicate that checks treeness and orderness together in another context where we need only treeness and have no node values. (In this specific example, `repOk` would not even need to check treeness if it maintains a set of visited values; checking that the values are ordered and not repeated would imply treeness.)

Ineffective Properties

We next discuss properties for which there is no imperative predicate that Korat can efficiently solve. In this discussion, we consider the number of invalid candidate structures as a measure of Korat efficiency. We do not consider the time that Korat takes; despite the large number of invalid candidate structures, Korat may still be able to generate in reasonable time all nonisomorphic valid structures, within small scopes, for these properties. We also do not consider the number of valid structures: it may be very large, but it depends on the property, and Korat cannot control it; for example, for predicate `true`, all structures are valid. We focus on the efficiency of Korat's pruning.

Korat is ineffective for every global property that (1) depends on a large number of fields, usually the whole structure; and (2) is not expressible as a conjunction of local properties that depend on a smaller number of fields. Consider, for example, the property that requires the sum of array elements to be equal to some constant. This property depends on the whole array and cannot be checked locally, so Korat's pruning is ineffective for this property. In contrast, consider the property that requires all array elements to be different. This property also depends on the whole array, but we can check it with several more local properties: the second element is different from the first, the third element is different from the second and first, and so on.

Korat is ineffective for many numerical properties that depend on the whole structure. On the other hand, Korat is effective for the usual structural properties; although they globally depend on the whole structure (e.g., that a graph is a tree), we can check them with several more local properties (e.g., there are no local cycles). We hypothesize that structural properties are amenable to local checking due to the recursive nature of data structures: reasoning about data structures typically proceeds by structural induction where all substructures satisfy the same invariants.

11.1.3 Bounded-Exhaustive Testing

Bounded-exhaustive testing checks the code for all nonisomorphic inputs within a given scope. Our experiments show that this form of testing is very effective for data-structure benchmarks; it is feasible to obtain test suites that have high statement, branch, and mutation coverage. The use of bounded-exhaustive testing also discovered subtle faults in several real applications, including a naming architecture for dynamic networks [73], a constraint solver for first-order logic formulas [73], a fault-tree analyzer [104], web-service protocols, and XML processing tools [102].

It is clear, however, that bounded-exhaustive testing cannot detect all faults. For example, it will miss all software problems, be they correctness or performance, that are detected only with large inputs. It is also not applicable for finding concurrency problems such as race conditions and deadlocks. It is less applicable when test inputs are simple, say just an integer for a procedure that finds a square root, or when test inputs have to satisfy simple properties that almost all inputs from the input space satisfy. Bounded-exhaustive testing is the most effective for test inputs that are structurally complex and have to satisfy strong properties. This is the case for applications that manipulate complex, linked data structures; our solver Korat indeed focuses on generating such complex data.

A primary challenge in using bounded-exhaustive testing is the huge number of test inputs. While Korat may effectively generate these inputs, it may be prohibitively expensive to test the software for all of them. Actually, the scalability of bounded-exhaustive testing is limited by the complexity of data that the code operates on (and thus the potential number of inputs) rather than by the complexity and size of the code under test. For example, Sullivan et al. [104] used bounded-exhaustive testing for testing an application that has over 30,000 lines of C++ code [104], and testers at Microsoft used bounded-exhaustive testing for testing an application that has over 10,000 lines of C# code [102]. While it was feasible to test such code for millions of inputs, it may be infeasible to do so for billions.

Another challenge in bounded-exhaustive testing, as in many other techniques that automate test-input generation, is when to stop the generation. The best rule we can give is: stop when the testing becomes too slow, i,e., generate test suites increasing the scope until there are too many inputs. The user may additionally measure a certain coverage that the test inputs achieve. Our evaluation uses structural code coverage and mutation coverage. While measuring structural code coverage may be practical for any application, mutation testing may not be practical for applications with a large number of mutants, because of the large number of test inputs.

11.2 Future Work

Our test-generation tool requires the user to provide an imperative predicate. In Java, an imperative predicate is a method that takes an input structure and returns a boolean. Additionally, the user provides a finitization that bounds the number of objects in the structure and the values for the fields of those objects. Korat then generates all nonisomorphic structures, within the bounds, for which the predicate returns true. Two structures are isomorphic if they differ only in the identity of objects (Section 3.2.5). The structures that Korat generates form a basis for bounded-exhaustive testing. We next present several ideas for extending the work on Korat by changing the language for predicates, considering different types of generation, improving the performance of generation, and applying Korat in other areas.

Language Design: A key issue in generating test inputs from constraints such as imperative or declarative predicates is to find the right trade-off between the ease of writing the constraints and the efficiency of generation. Korat in its basic form already provides a good balance. The library of dedicated generators (Section 5.3) further allows a combination of predicates, which are often easier to write, and generators, which are often more efficient. However, to extend the library, the user needs to understand the internals of Korat. It would be beneficial to explore ways that make it easier for the user to provide specialized generators. We can facilitate this by adding backtracking to the languages that do not have it, such as Java or C#. It would be also interesting to explore how languages with different paradigms, such as Java and Alloy, can combine into one language for writing constraints on test inputs.

Nonequivalent Structures: Korat has a very efficient, built-in support for generating only nonisomorphic structures. To reduce the number of generated structures, however, the user may want to obtain only *nonequivalent* structures, with some equivalence weaker than isomorphism (Section 1.1.3). For instance, recall our `SearchTree` example: the user could consider two `SearchTree` objects equivalent if they have the same elements (regardless of the structure of the tree) or if they have the same structure (regardless of the elements in the tree). This arises in integration testing: once we have tested the `SearchTree` class, we want to use it at the abstract level, without considering the concrete data representation that the implementation uses. In such context, it is redundant to test with equivalent structure as test inputs [116].

There are basically two approaches to describe equivalent structures [116]: (1) provide an abstraction function that maps each structure into an element of some other domain, for example trees into sets, in which case two structures are equivalent iff they map to the same element; or (2) provide a binary predicate that takes two structures and returns true or false, for example, `equals` methods in Java. A straightforward approach to generate only nonequivalent structures is to generate all (nonisomorphic) structures and then filter out the equivalent ones. However, the question is if we can eliminate nonequivalent structures more efficiently during the generation, akin to how Korat eliminates nonisomorphic structures. One technique to consider is to have specialized nonequivalence generators for a certain type of structures and then only allow the abstraction functions that map into those structures.

Nonexhaustive Generation: An obvious approach to reduce the number of structures is to consider nonexhaustive generation. Instead of generating all structures, we could, for example, randomly generate some of them. We can easily adapt Korat for nonexhaustive search: instead of generating candidate structures following the usual strategy, we could generate them with another strategy. For example, we could randomly choose some candidate, then apply Korat search until it finds a valid structure following the candidate, and then again randomly choose another candidate. Korat is also amenable for applications of simulated annealing or genetic algorithms [92].

In particular, we think that *neighborhood exploration* would be an interesting globally nonexhaustive, but locally exhaustive, strategy for generating structures. Neighborhood exploration generates all structures similar (according to an appropriate metric) to some given structure, for example, trees that have at most one node more or less than the given tree. This strategy could be especially useful in the following testing scenario: (1) run a deployed program and record the actual inputs to some unit under test; (2) automatically generate all inputs similar to the recorded inputs and test the program for them. This would enable focusing on input patterns that are important in practice.

Faster Solving: Given a predicate, Korat generates valid structures by searching through the predicate's structure space; Korat prunes the search by monitoring the executions of the predicate on candidate structures and keeping track of the accessed fields. This technique works well in practice, but in theory monitoring only field accesses is imprecise. We could instead keep track of more refined *dynamic dependencies* to detect which fields the return value actually depends on: even though an execution reads some field, the return value may have no (data or control) dependency on that field. This approach would make executions of candidates slower than in Korat, due to additional book keeping, but it could reduce the number of candidates. Without an empirical evaluation on actual imperative predicates, it is hard to compare this approach with Korat.

Instead of finding only which fields the return value depends on, we could use *symbolic execution* to find *how* the return value depends on the values of those fields. We could then use *constraint solving* to find the values for the fields that make the return value true. Khurshid et al. [58,110] have started exploring this approach with a symbolic execution that uses symbolic expressions for primitive values and concrete pointers for object references. The initial results on two benchmarks are promising, but it is unclear hot they generalize.

Predicate Synthesis: Currently Korat requires the user to provide an imperative predicate that characterizes test inputs. We could consider automatic synthesis of predicates from the code under test and/or from the specific test inputs that the user provides. Pacheco and Ernst have recently developed Eclat [88], a tool that automatically generates additional test inputs from a given set of test inputs and the code under test. Eclat builds on Daikon [31], a tool that dynamically detects likely program invariants, to generalize likely predicates from the given tests. Eclat uses random generation, and we could instead use Korat for bounded-exhaustive generation.

Novel Applications: There may be several other applications for solvers for imperative predicates besides bounded-exhaustive testing. We discuss a potential application in enumerative combinatorics. Researchers in this area often write programs that count objects of interest by generating them for different sizes. Researchers then analyze the obtained sequences. Counting objects by generation corresponds to bounded-exhaustive generation. Optimizing counting programs can take more effort than the benefit gained from obtaining a longer sequence [74]. To make it easier to generate combinatorial objects, we could (1) design a domain-specific language for specifying combinatorial properties (the language should build on mathematical notation familiar to combinatorics researchers) and (2) develop a Korat-like tool that generates objects with given properties by combining highly-optimized, user-contributed specialized generators and a general Korat-like search.

11.3 Summary

Despite the recent advances in verification and validation technology, testing remains the most widely used method for increasing software reliability. Moreover, the importance of testing is increasing as the consequences of software errors become more severe. Software nowadays controls numerous embedded and infrastructural systems where errors can lead to great financial losses or even loss of life. Software errors can also lead to security and privacy vulnerabilities, and we are witnessing a decreased tolerance for errors even from the end users of noncritical applications.

At the same time, software systems are increasing in complexity. Software is becoming more connected; seemingly benign errors in one part of the system can easily propagate to the other parts and cause serious problems. As a result, testing is gaining importance and software developers are spending more time testing. Recent software development methodologies, such as Extreme Programming [8]—also known as test-driven development or test-first programming, require developers to develop tests even before developing the code.

We focus on software that manipulates structurally complex data. Such software is increasingly pervasive in modern systems and poses important challenges for testing. In particular, manually developing test suites for such code is very tedious. Automated testing can significantly help programmers to develop and maintain reliable software.

This dissertation presented an approach for testing based on automatic generation of structurally complex test inputs from imperative predicates—pieces of code that check the properties of the desired test inputs. Our main contribution is a technique for solving imperative predicates and an implementation of this technique in a tool called Korat. Given a predicate and a bound on its inputs, Korat automatically generates all inputs that satisfy the properties. Testing the code with all such inputs corresponds to bounded-exhaustive testing.

This dissertation presented results that show that bounded-exhaustive testing is effective for unit testing of data-structure implementations. It is feasible to use Korat to generate all test inputs within small bounds. The key to efficient generation are the optimizations that Korat applies: it prunes a vast majority of invalid test inputs and eliminates isomorphic test inputs. The Korat technique for solving imperative predicates has also been adopted in industry, where its use found previously unknown errors in important real-world applications that were already well tested. These results suggest that bounded-exhaustive testing is a practical way for testing applications that manipulate complex data.

Appendix A

Full Example Code and Experimental Results

This appendix presents the full code for the `SearchTree` example used throughout the dissertation and the full experimental results for method testing described in Section 8.3. Figures A-1 and A-2 show the full code for the example `remove` and `repOk` methods introduced in Section 2.1. Figure A-3 presents detailed experimental results for bounded-exhaustive testing of our data structure benchmarks.

```
boolean remove(int info) {
    Node parent = null;
    Node current = root;
    while (current != null) {
        if (info < current.info) {
            parent = current;
            current = current.left;
        } else if (info > current.info) {
            parent = current;
            current = current.right;
        } else {
            break;
        }
    }
    if (current == null) return false;
    Node change = removeNode(current);
    if (parent == null) {
        root = change;
    } else if (parent.left == current) {
        parent.left = change;
    } else {
        parent.right = change;
    }
    return true;
}
```

```
Node removeNode(Node current) {
    size--;
    Node left = current.left;
    Node right = current.right;
    if (left == null) return right;
    if (right == null) return left;
    if (left.right == null) {
        current.info = left.info;
        current.left = left.left;
        return current;
    }
    Node temp = left;
    while (temp.right.right != null) {
        temp = temp.right;
    }
    current.info = temp.right.info;
    temp.right = temp.right.left;
    return current;
}
```

Figure A-1: Full code for the example `remove` method.

```
boolean repOk() {
    // checks that empty tree has size zero
    if (root == null) return size == 0;
    // checks that the object graph is a tree
    if (!isTree()) return false;
    // checks that size is consistent
    if (numNodes(root) != size) return false;
    // checks that data is ordered
    if (!isOrdered(root)) return false;
    return true;
}

/** Checks that the input object graph
    is actually a tree.
    Uses a work-list based
    breadth-first traversal. */
boolean isTree() {
    Set visited = new HashSet();
    visited.add(root);
    LinkedList workList = new LinkedList();
    workList.add(root);
    while (!workList.isEmpty()) {
        Node current =
            (Node)workList.removeFirst();
        if (current.left != null) {
            // checks that the graph
            // has no sharing
            if (!visited.add(current.left))
                return false;
            workList.add(current.left);
        }
        if (current.right != null) {
            // checks that the graph
            // has no sharing
            if (!visited.add(current.right))
                return false;
            workList.add(current.right);
        }
    }
    return true;
}
```

```
/** Returns the number of the nodes
    in the tree. */
//@ requires isTree();
int numNodes(Node n) {
    if (n == null) return 0;
    return 1 + numNodes(n.left) +
           numNodes(n.right);
}

/** Checks that the subtree
    rooted in n is ordered. */
//@ requires isTree();
boolean isOrdered(Node n) {
    return isOrdered(n, Integer.MIN_VALUE,
                        Integer.MAX_VALUE);
}

/** Checks that all values
    in the subtree rooted in n
    are between the bounds
    min and max, inclusive. */
boolean isOrdered(Node n,
                  int min, int max) {
    if ((n.info < min) ||
        (n.info > max))
        return false;
    if (n.left != null) {
        if (n.info == min)
            return false;
        if (!isOrdered(n.left, min,
                       n.info - 1))
            return false;
    }
    if (n.right != null) {
        if (n.info == max)
            return false;
        if (!isOrdered(n.right,
                       n.info + 1, max))
            return false;
    }
    return true;
}
```

Figure A-2: Full code for the example repOk method.

		generation					testing			
		gen.	ded.	spec. coverage		#	time	code coverage		m.c.
benchmark	scope	[sec]	[sec]	st. [%]	br. [%]	inputs	[sec]	st. [%]	br. [%]	[%]
SearchTree	1	0.06	0.01	57.89	60.00	4	0.06	38.46	40.00	26.10
	2	0.05	0.01	94.74	96.67	20	0.06	79.49	87.50	69.85
	3	0.07	0.10	94.74	96.67	90	0.07	87.18	92.50	79.77
	4	0.17	0.10	94.74	96.67	408	0.14	97.44	97.50	92.64
	5	0.38	0.25	94.74	96.67	1880	0.24	100.00	100.00	98.52
	6	1.39	0.52	94.74	96.67	8772	0.46	100.00	100.00	99.26
	7	9.03	2.19	94.74	96.67	41300	1.25	100.00	100.00	99.26
DisjSet	1	0.01	0.01	61.54	55.00	4	0.04	23.08	25.00	0.41
	2	0.01	0.01	100.00	95.00	30	0.09	69.23	68.75	30.45
	3	0.04	0.04	100.00	100.00	456	0.09	100.00	100.00	88.47
	4	0.29	0.31	100.00	100.00	18280	0.43	100.00	100.00	95.06
	5	10.91	9.87	100.00	100.00	1246380	19.93	100.00	100.00	95.06
HeapArray	1	0.01	0.01	80.00	85.71	16	0.04	79.31	66.67	39.05
	2	0.01	0.01	90.00	92.86	75	0.05	79.31	66.67	43.79
	3	0.02	0.02	90.00	92.86	396	0.09	93.10	83.33	69.70
	4	0.08	0.09	90.00	92.86	2240	0.17	96.55	88.89	86.13
	5	0.22	0.21	90.00	92.86	15352	0.38	96.55	94.44	89.78
	6	0.90	0.71	90.00	92.86	118251	1.88	100.00	100.00	96.35
	7	7.09	6.21	90.00	92.86	1175620	17.58	100.00	100.00	96.71
BinHeap	1	0.02	0.01	62.79	62.00	12	0.07	52.87	57.58	31.16
	2	0.03	0.02	93.02	94.00	54	0.08	87.36	84.85	62.67
	3	0.12	0.09	93.02	94.00	336	0.14	98.85	96.97	89.72
	4	0.40	0.30	97.67	98.00	1800	0.24	100.00	98.48	93.15
	5	0.81	0.65	97.67	98.00	16848	0.69	100.00	100.00	94.86
	6	3.30	2.35	97.67	98.00	159642	4.61	100.00	100.00	95.89
	7	35.60	28.06	97.67	98.00	2577984	75.96	100.00	100.00	96.91
FibHeap	1	0.01	0.07	55.55	51.72	12	0.07	35.48	43.55	15.82
	2	0.03	0.03	91.11	93.10	108	0.09	75.27	80.64	44.10
	3	0.28	0.24	97.78	98.28	1632	0.24	95.70	98.39	75.08
	4	1.22	0.90	97.78	98.28	34650	1.08	95.70	98.39	81.48
	5	14.14	12.94	97.78	98.28	941058	23.37	100.00	100.00	88.88
LinkedList	1	0.01	0.01	100.00	100.00	15	0.08	64.15	68.75	58.19
	2	0.01	0.01	100.00	100.00	50	0.09	90.57	84.38	98.77
	3	0.03	0.03	100.00	100.00	169	0.12	90.57	84.38	99.59
	4	0.07	0.07	100.00	100.00	627	0.16	90.57	84.38	99.59
	5	0.18	0.18	100.00	100.00	2584	0.26	90.57	84.38	99.59
	6	0.33	0.31	100.00	100.00	11741	0.48	90.57	84.38	99.59
	7	0.74	0.71	100.00	100.00	58175	1.54	90.57	84.38	99.59
SortedList	1	0.03	0.04	71.43	62.50	7	0.11	62.50	50.00	33.33
	2	0.04	0.07	100.00	100.00	36	0.11	80.00	74.14	52.81
	3	0.07	0.07	100.00	100.00	188	0.15	92.50	89.66	90.04
	4	0.22	0.20	100.00	100.00	1066	0.28	92.50	89.66	93.93
	5	0.53	0.48	100.00	100.00	7427	0.50	92.50	89.66	96.53
	6	1.94	1.77	100.00	100.00	73263	2.57	92.50	89.66	97.40
	7	22.68	21.13	100.00	100.00	1047608	37.91	92.50	89.66	97.40
TreeMap	1	0.02	0.02	57.14	63.33	6	0.06	14.41	14.89	5.46
	2	0.03	0.03	100.00	100.00	28	0.06	45.95	50.00	28.66
	3	0.07	0.04	100.00	100.00	96	0.09	63.96	73.40	61.09
	4	0.18	0.15	100.00	100.00	328	0.15	89.19	85.11	78.15
	5	0.38	0.31	100.00	100.00	1150	0.24	100.00	91.49	87.37
	6	0.94	0.61	100.00	100.00	3924	0.38	100.00	91.49	89.76
	7	3.28	1.75	100.00	100.00	12754	0.73	100.00	91.49	89.76
HashSet	1	0.01	0.01	57.89	69.23	4	0.04	51.92	50.00	29.91
	2	0.01	0.01	89.47	92.31	34	0.05	96.15	95.00	77.45
	3	0.06	0.05	89.47	92.31	212	0.09	100.00	100.00	90.57
	4	0.23	0.22	89.47	92.31	1170	0.19	100.00	100.00	90.98
	5	0.36	0.34	89.47	92.31	3638	0.27	100.00	100.00	91.39
	6	0.91	0.71	89.47	92.31	12932	0.62	100.00	100.00	91.80
	7	3.38	2.88	89.47	92.31	54844	1.55	100.00	100.00	92.21
AVTree	1	0.01	0.01	53.33	56.25	2	0.07	55.29	51.92	40.00
	2	0.05	0.03	90.00	87.50	86	0.14	75.29	78.85	60.00
	3	0.21	0.17	96.67	96.88	1702	0.78	88.23	84.61	75.12
	4	3.16	1.86	96.67	96.88	27734	8.36	94.12	92.31	91.21
	5	87.13	43.41	96.67	96.88	417878	134.51	94.12	92.31	93.65

Figure A-3: Korat's performance for test generation and checking. We tabulate the time for test generation (with regular and dedicated generators), specification coverage (statement and branch), the time for correctness checking, code coverage (statement and branch), and mutation coverage (the percentage of killed mutants). All times are elapsed real times in seconds from the start of Korat to its completion.

Bibliography

[1] William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. In *Proc. 17th ACM Symposium on Operating Systems Principles (SOSP)*, Kiawah Island, December 1999.

[2] Hiralal Agrawal, Richard A. DeMillo, R. Hathaway, William Hsu, Wynne Hsu, Edward W. Krauser, Rhonda J. Martin, Aditya P. Mathur, and Eugene H. Spafford. Design of mutant operators for the C programming language. Technical Report SERC-TR-41-P, Purdue University, West Lafayette, IN, 1989.

[3] Fadi A. Aloul, Arathi Ramani, Igor L. Markov, and Karem A. Sakallah. Solving difficult SAT instances in the presence of symmetry. In *Proc. 39th Conference on Design Automation*, pages 731–736, 2002.

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

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

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

[7] Roberto J. Bayardo Jr. and Robert C. Schrag. Using CSP look-back techniques to solve real world SAT instances. In *Proc. National Conference on Artificial Intelligence*, pages 203–208, 1997.

[8] Kent Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley, 2000.

[9] Kent Beck and Erich Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7), July 1998.

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

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

[12] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshun Zhu. *Bounded Model Checking*, volume 58 of *Advances in Computers.* Elsevier, 2003.

[13] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, pages 123–133, July 2002.

[14] Lilian Burdy, Yoonsik Cheon, David Cok, Michael D. Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *Software Tools for Technology Transfer*, 2005. (to appear).

[15] Mattias Bybro. A mutation testing tool for java programs. Master's thesis, Stockholm University, Stockholm, Sweden, 2003.

[16] Juei Chang and Debra J. Richardson. Structural specification-based testing: Automated support and experimental evaluation. In *Proc. 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 285–302, September 1999.

[17] Yoonsik Cheon and Gary T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *Proc. European Conference on Object-Oriented Programming (ECOOP)*, June 2002.

[18] Philippe Chevalley and Pascale Thévenod-Fosse. A mutation analysis tool for Java programs. *Journal on Software Tools for Technology Transfer (STTT)*, pages 1–14, December 2002.

[19] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of Haskell programs. In *Proc. Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 268–279, 2000.

[20] Koen Claessen and John Hughes. Testing monadic code with QuickCheck. In *Proc. ACM SIGPLAN workshop on Haskell*, pages 65–77, 2002.

[21] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton. The AETG system: An approach to testing based on combinatorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, 1997.

[22] James Corbett, Matthew Dwyer, John Hatcliff, Corina Pasareanu, Robby, Shawn Laubach, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. 22nd International Conference on Software Engineering (ICSE)*, June 2000.

[23] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, MA, 1990.

[24] James Crawford, Matthew Ginsberg, Eugene Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In *Proc. Fifth International Conference on Principles of Knowledge Representation and Reasoning*, 1996.

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

[26] Marcio E. Delamaro and Jose C. Maldonado. Proteum—A tool for the assessment of test adequacy for C programs. In *Conference on Performability in Computing Systems (PCS 96)*, New Brunswick, NJ, July 1996.

[27] Claudio Demartini, Radu Iosif, and Riccardo Sisto. A deadlock detection tool for concurrent Java programs. *Software Practice and Experience*, July 1999.

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

[29] Richard A. DeMillo and A. Jefferson Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9), 1991.

[30] Michael R. Donat. Automating formal specification based testing. In *Proc. Conference on Theory and Practice of Software Development*, volume 1214, pages 833–847, Lille, France, 1997.

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

[32] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proc. ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, 2002.

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

[34] Phyllis Frankl, Stewart Weiss, and Cang Hu. All uses versus mutation testing: An experimental comparison of effectiveness. *Journal of Systems and Software*, 38(3):235–253, 1997.

[35] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements od Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison-Wesley Publishing Company, New York, NY, 1995.

[36] Patrice Godefroid. Model checking for programming languages using VeriSoft. In *Proc. 24th Annual ACM Symposium on the Principles of Programming Languages (POPL)*, pages 174–186, Paris, France, January 1997.

[37] John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, June 1975.

[38] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification.* Sun Microsystems, Inc., 2001.

[39] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. Automatic test data generation using constraint solving techniques. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, Clearwater Beach, FL, 1998.

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

[41] Ralph E. Griswold and Madge T. Griswold. *The Icon programming language.* Peer-to-Peer Communications, San Jose, CA, third edition, 1996.

[42] Yuri Gurevich. Evolving algebras 1993: Lipari guide. In *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

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

[44] Richard G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4), July 1977.

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

[46] Gerald Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

[47] Hans-Martin Horcher. Improving software tests using Z specifications. In *Proc. 9th International Conference of Z Users, The Z Formal Specification Notation*, 1995.

[48] W.E. Howden. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, SE-2(3):208–215, 1976.

[49] William E. Howden. Symbolic testing and the dissect symbolic evaluation system. *IEEE Transactions on Software Engineering*, 3(4):266–278, 1977.

[50] J. C. Huang. An approach to program testing. *ACM Computing Surveys*, 7(3), 1975.

[51] Daniel Jackson, Ian Schechter, and Ilya Shlyakhter. ALCOA: The Alloy constraint analyzer. In *Proc. 22nd International Conference on Software Engineering (ICSE)*, Limerick, Ireland, June 2000.

[52] Daniel Jackson, Ilya Shlyakhter, and Manu Sridharan. A micromodularity mechanism. In *Proc. 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Vienna, Austria, September 2001.

[53] Daniel Jackson and Mandana Vaziri. Finding bugs with a constraint solver. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, Portland, OR, August 2000.

[54] Bart P. F. Jacobs, Joseph R. Kinirya, and Martijn E. Warnier. Java program verification challenges. Technical Report NIII-R0310, Computing Science Institute, University of Nijmegen, 2003.

[55] Sarfraz Khurshid. *Generating Structurally Complex Tests from Declarative Constraints*. PhD thesis, Dept. of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, December 2003.

[56] Sarfraz Khurshid and Darko Marinov. Checking Java implementation of a naming architecture using TestEra. In Scott D. Stoller and Willem Visser, editors, *Electronic Notes in Theoretical Computer Science (ENTCS)*, volume 55. Elsevier Science Publishers, 2001.

[57] Sarfraz Khurshid, Darko Marinov, Ilya Shlyakhter, and Daniel Jackson. A case for efficient solution enumeration. In *Proc. Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT 2003)*, Santa Margherita Ligure, Italy, May 2003.

[58] Sarfraz Khurshid, Corina Pasareanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *Proc. 9th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, Warsaw, Poland, April 2003.

[59] Sun-Woo Kim, John Clark, and John McDermid. The rigorous generation of Java mutation operators using HAZOP. In *12th International Conference on Software & Systems Engineering and their Applications (ICSSEA'99)*, December 1999.

[60] Sun-Woo Kim, John Clark, and John McDermid. Class mutation: Mutation testing for object oriented programs. In *FMES 2000*, October 2000.

[61] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[62] K. N. King and A. Jefferson Offutt. A Fortran language system for mutation-based software testing. *Software Practice and Experience*, 21(7):685–718, 1991.

[63] Bogdan Korel. Automated test data generation for programs with procedures. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, San Diego, CA, 1996.

[64] Viktor Kuncak, Patrick Lam, and Martin Rinard. Role analysis. In *Proc. 29th Annual ACM Symposium on the Principles of Programming Languages (POPL)*, Portland, OR, January 2002.

[65] Patrick Lam, Viktor Kuncak, and Martin Rinard. Generalized typestate checking for data structure consistency. In *Proc. 6th International Conference on Verification, Model Checking and Abstract Interpretation*, January 2005. (to appear).

[66] Butler W. Lampson. Principles of computer systems. `http://web.mit.edu/6.826/www/notes`.

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

[68] Bruno Legeard, Fabien Peureux, and Mark Utting. Automated boundary testing from Z and B. In *Proc. Formal Methods Europe (FME)*, Copenhagen, Denmark, July 2002.

[69] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison Wesley, second edition, 1999.

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

[71] Yu-Seung Ma, Yong-Rae Kwon, and Jeff Offutt. Inter-class mutation operators for Java. In *Proc. 13th International Symposium on Software Reliability Engineering*, pages 352–363, 2002.

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

[73] Darko Marinov and Sarfraz Khurshid. TestEra: A novel framework for automated testing of Java programs. In *Proc. 16th IEEE International Conference on Automated Software Engineering (ASE)*, San Diego, CA, November 2001.

[74] Darko Marinov and Radoš Radoičić. Counting 1324-avoiding permutations. *Electronic Journal of Combinatorics*, 9(2), May 2003.

[75] Peter M. Maurer. Generating test data with enhanced context-free grammars. *IEEE Software*, 7(4):50–55, July 1990.

[76] William M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

[77] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[78] Laurent Michel and Pascal Van Hentenryck. A constraint-based architecture for local search. In *Proc. ACM SIGPLAN 2002 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 83–100, 2002.

[79] Francesmary Modugno, Albert T. Corbett, and Brad A. Myers. Graphical representation of programs in a demonstrational visual shell—An empirical evaluation. *ACM Transactions on Computer-Human Interaction*, 4(3):276–308, 1997.

[80] Anders Moeller and Michael I. Schwartzbach. The pointer assertion logic engine. In *Proc. SIGPLAN Conference on Programming Languages Design and Implementation*, Snowbird, UT, June 2001.

[81] Ivan Moore. Jester—A JUnit test tester. In *2nd International Conference on eXtreme Programming and Flexible Processes in Software Engineering*, May 2001.

[82] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 39th Design Automation Conference (DAC)*, June 2001.

[83] Madanlal Musuvathi, David Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *Proc. Fifth Symposium on Operating Systems Design and Implementation*, December 2002.

[84] Jeff Offutt and Aynur Abdurazik. Generating tests from UML specifications. In *Proc. Second International Conference on the Unified Modeling Language*, October 1999.

[85] Jeff Offutt, Yu-Seung Ma, and Yong-Rae Kwon. An experimental mutation system for Java. *ACM SIGSOFT Software Engineering Notes*, 29(5):1–4, 2004.

[86] Jeff Offutt and Roland Untch. Mutation 2000: Uniting the orthogonal. In *Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, San Jose, CA, October 2000.

[87] Jeff Offutt, Jeff Voas, and Jeff Payne. Mutation operators for Ada. Technical Report ISSE-TR-96-09, George Mason University, Fairfax, VA, October 1996.

[88] Carlos Pacheco and Michael D. Ernst. Eclat: Automatic generation and classification of test inputs. Technical Report MIT-LCS-TR-968, MIT CSAIL, Cambridge, MA, October 2004.

[89] Parasoft. Jtest version 5.1. Online manual, July 2004. `http://www.parasoft.com/`.

[90] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: An extensible and highly-modular software model checking framework. In *Proc. ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 267–276, September 2003.

[91] Radu Rugina. Quantitative shape analysis. In *Proc. 11th International Static Analysis Symposium (SAS'04)*, August 2004.

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

[93] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, January 1998.

[94] Sriram Sankar and Roger Hayes. Specifying and testing software components using ADL. Technical Report SMLI TR-94-23, Sun Microsystems Laboratories, Inc., Mountain View, CA, April 1994.

[95] Jacob T. Schwartz, Robert B. Dewar, Edmond Schonberg, and Ed Dubinsky. *Programming with Sets: An Introduction to SETL.* Springer-Verlag, New York, NY, 1986.

[96] Ilya Shlyakhter. Generating effective symmetry-breaking predicates for search problems. In *Proc. Workshop on Theory and Applications of Satisfiability Testing*, June 2001.

[97] Emin Gün Sirer and Brian N. Bershad. Using production grammars in software testing. In *Proc. 2nd conference on Domain-specific languages*, pages 1–13, 1999.

[98] N. J. A. Sloane, Simon Plouffe, J. M. Borwein, and Robert M. Corless. The encyclopedia of integer sequences. *SIAM Review*, 38(2), 1996. `http://www.research.att.com/~njas/sequences/Seis.html`.

[99] Fabio Somenzi. CUDD: CU decision diagram package. `http://vlsi.colorado.edu/~fabio/CUDD/`.

[100] J. Mike Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, second edition, 1992.

[101] Richard P. Stanley. *Enumerative combinatorics. Vol. 1.* Cambridge University Press, New York, NY, 1997.

[102] Keith Stobie. Advanced modeling, model based test generation, and Abstract state machine Language (AsmL). Seattle Area Software Quality Assurance Group, `http://www.sasqag.org/pastmeetings/asml.ppt`, January 2003.

[103] Phil Stocks and David Carrington. A framework for specification-based testing. *IEEE Transactions on Software Engineering*, 22(11):777–793, 1996.

[104] Kevin Sullivan, Jinlin Yang, David Coppit, Sarfraz Khurshid, and Daniel Jackson. Software assurance by bounded exhaustive testing. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, pages 133–142, 2004.

[105] Sun Microsystems. *Java 2 Platform, Standard Edition, v1.3.1 API Specification.* `http://java.sun.com/j2se/1.3/docs/api/`.

[106] Roland Untch, A. Jefferson Offutt, and Mary Jean Harrold. Mutation testing using mutant schemata. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*, pages 139–148, June 1993.

[107] Mandana Vaziri. *Finding Bugs Using a Constraint Solver.* PhD thesis, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 2003.

[108] Mandana Vaziri and Daniel Jackson. Checking properties of heap-manipulating procedures with a constraint solver. In *Proc. 9th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, Warsaw, Poland, April 2003.

[109] Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model checking programs. In *Proc. 15th IEEE International Conference on Automated Software Engineering (ASE)*, Grenoble, France, 2000.

[110] Willem Visser, Corina S. Pasareanu, and Sarfraz Khurshid. Test input generation with Java PathFinder. In *Proc. 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 97–107, 2004.

[111] Patrick J. Walsh. *A Measure of Test Case Completeness.* PhD thesis, State University of New York at Binghamton, 1985.

[112] Elaine J. Weyuker and Thomas J. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Transactions on Software Engineering*, 6(3):236–246, May 1980.

[113] W. Eric Wong. *On Mutation and Data Flow.* PhD thesis, Purdue University, West Lafayette, IN, 1993.

[114] World Wide Web Consortium (W3C). Web services activity. `http://www.w3.org/2002/ws`.

[115] World Wide Web Consortium (W3C). XML path language (XPath). `http://www.w3.org/TR/xpath`.

[116] Tao Xie, Darko Marinov, and David Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *Proc. 19th IEEE International Conference on Automated Software Engineering*, pages 196–205, September 2004.

[117] Guoqing Xu and Zongyuan Yang. JMLAutoTest: A novel automated testing framework based on JML and JUnit. In *Proc. 3rd International Workshop on Formal Approaches to Testing of Software (FATES)*, Montreal, Canada, October 2003.

[118] Greta Yorsh, Thomas W. Reps, and Shmuel Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 530–545, 2004.

[119] Karen Zee, Patrick Lam, Viktor Kuncak, and Martin Rinard. Combining theorem proving with static analysis for data structure consistency. In *International Workshop on Software Verification and Validation (SVV 2004)*, Seattle, November 2004.

[120] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.