# Generating Structurally Complex Tests from Declarative Constraints

by

Sarfraz Khurshid

S.M., Massachusetts Institute of Technology (2000)
Part III of the Mathematical Tripos, Trinity College Cambridge (1998)
B.Sc. Honours, Imperial College London (1997)
B.Sc., Government College Lahore (1993)

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

Dec 2003

Author . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Department of Electrical Engineering and Computer Science
Dec 17, 2003

Certified by . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
Daniel Jackson
Associate Professor
Thesis Supervisor

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

# Generating Structurally Complex Tests from Declarative Constraints

by

Sarfraz Khurshid

Submitted to the Department of Electrical Engineering and Computer Science on Dec 17, 2003, in partial fulfillment of the requirements for the degree of Doctor of Philosophy

## Abstract

This dissertation describes a method for systematic constraint-based test generation for programs that take as inputs structurally complex data, presents an automated SAT-based framework for testing such programs, and provides evidence on the feasibility of using this approach to generate high quality test suites and find bugs in non-trivial programs.

The framework tests a program systematically on all nonisomorphic inputs (within a given bound on the input size). Test inputs are automatically generated from a given input constraint that characterizes allowed program inputs. In unit testing of object-oriented programs, for example, an input constraint corresponds to the representation invariant; the test inputs are then objects on which to invoke a method under test. Input constraints may additionally describe test purposes and test selection criteria.

Constraints are expressed in a simple (first-order) relational logic and solved by translating them into propositional formulas that are handed to an off-the-shelf SAT solver. Solutions found by the SAT solver are lifted back to the relational domain and reified as tests.

The TestEra tool implements this framework for testing Java programs. Experiments on generating several complex structures indicate the feasibility of using off-the-shelf SAT solvers for systematic generation of nonisomorphic structures. The tool also uncovered previously unknown errors in several applications including an intentional naming scheme for dynamic networks and a fault-tree analysis system developed for NASA.

Thesis Supervisor: Daniel Jackson
Title: Associate Professor

# Acknowledgments

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 Problem Description and Proposed Solution

Software testing, the most commonly used technique for validating the quality of software, is a labor intensive process, and typically accounts for about half the total cost of software development and maintenance [10]. Automating testing would not only reduce the cost of producing software but also increase the reliability of modern software. A recent report by the National Institute of Standards and Technology estimates that software failures currently cost the US economy about $60 billion every year, and that improvements in software testing infrastructure might save one-third of this cost [2].

One might question why there is such room for improvement at all, given the conceptually simple nature of testing: just create a *test suite*, i.e., a set of test inputs, run them against the program, and check if each output is correct. The key issue with the current practice of testing is the need to manually generate test suites. Generating an input manually requires considerable effort that is often wasted: executing a program on an input and not detecting a bug does not help correct potential errors. Sometimes the effort is so significant that developers resort to using a very small and inadequate set of inputs to test their applications, even if they are used in building mission critical systems. Indeed, such practice may result in erroneous applications being used for extended periods of time before subtle errors lead to failures.

### 1.1.1 Test Generation is Burdensome

So why do developers need to create suites manually, rather than generating them automatically? Generation would be straightforward if desired inputs were easy to sample, e.g., if they belonged to a given range of integers. For most programs, however, inputs need to satisfy certain complex properties. Consider, for example, a program that removes an element from a binary tree. For correct behavior the program may require that its input tree is in fact a *binary search tree*, with the elements in the tree appearing in the correct (search) order. As another example, consider testing a word processor GUI by generating an event sequence as an input. Certain sequences cannot be possible inputs, e.g., it may not be possible to change the font without first selecting the "Format" menu, and incorrect program behavior on such a sequence most likely amounts only to a false positive.

### 1.1.2 Key Idea: Generating Tests from Constraints

A key idea in our work is to generate tests from logical constraints. Even though manual input generation is laborious, describing properties of desired input data with an *input constraint* is typically simple. In object-oriented programs, such constraints are often already explicitly recorded in the code by developers. In particular, a representation invariant [63] (also called *class invariant*) constrains the representation of objects of the class and must be preserved by the operations that modify the representation. A method typically has a *precondition* that defines allowed inputs and a *postcondition* that states expected behavior; the class invariants implicitly forms a part of the precondition and postcondition. The input constraint for a method is its precondition.

An input constraint need not necessarily characterize the allowed program inputs. It is often the case that programmers want to test the robustness of their programs on inputs that violate certain properties; an input constraint then simply expresses the negation of these properties. An input constraint may also state a *test purpose*, such as that input trees should have at least three nodes, to focus testing of a particular behavior of interest.

A key advantage of using input constraints is that the constraints typically define a whole class of inputs and not just a small subset of that class. This enables using an appropriate constraint solver to enumerate an entire class using the same constraint. In contrast, a set of concrete inputs cannot (easily) be used to generate further inputs.

The use of constraints allows partiality in testing: a piece of code (or simply a method) can be taken in isolation from the rest of the implementation and tested by defining the constraints on program variables at the control point that represents start of execution. For example, the method that removes an element from a binary search tree can be tested without first having to implement the methods that build such trees.

To test a program, we systematically solve its input constraint to generate all inputs within a given bound on the input size, and test the program against the resulting suite.

### 1.1.3 The Challenge of Complex Structures

To determine how to solve input constraints, it is important first to understand the nature of the constraints. Prior techniques [19,42,56,74] for generating tests from constraints have considered constraints on primitive data, such as integers and booleans, and used dedicated solvers for such constraints.

In object-oriented software, data with complex *structure* are pervasive. Such data are defined by their structural constraints, e.g., in a binary tree, there are no directed cycles, each node has a unique parent and no node has the same node as both left and right child.

Solving structural constraints is not the only challenge in generating complex structures. Indeed, how to intuitively express these constraints by itself poses a challenge. Further, if the constraints are expressed in an abstract logic, translating solutions of constraints into actual tests poses another challenge.

### 1.1.4 Embodiment: The TestEra Tool

The TestEra tool presents an embodiment of how we address these challenges for automated testing of Java programs. The constraints are expressed in a simple (first-order) relational logic that allows the use of intuitive path expressions to build specifications; the constraints are solved by translating them into propositional formulas that are solved using an off-the-shelf SAT solver; the solutions found by the SAT solver are lifted back to the relational domain using a concretization translation and reified as tests.

Experiments on generating several complex structures indicate the feasibility of using off-the-shelf SAT solvers for systematic generation of nonisomorphic structures. The tool also uncovered previously unknown errors in several applications.

## 1.2 Example

To describe the problem domain and illustrate our framework, we introduce a simple example that we use throughout this chapter. Consider the Java code in Figure 1-1. The code declares a binary search tree and its `remove` method, which takes a tree, and an element, which it supposedly removes. The code of `remove` is shown in Appendix A.

Each object of the class `BinarySearchTree` represents a binary search tree; objects of the inner class `Node` represent nodes of the trees. The method `remove` has two inputs, i.e., the implicit `this` argument, which is a `BinarySearchTree`, and `i`, which is an integer. We denote an input to `remove` as the pair ($T$, $i$), where $T$ is the input tree and $i$ is the input element to delete.

Consider the following specification for this method: both precondition and postcondition require implicitly that `this` satisfy the class invariant for `BinarySearchTree`, which requires that the graph of nodes reachable from `root` indeed be a tree (i.e., have no cycles), the elements in the tree appear in the correct (search) order, and that the `size` field represents the number of nodes in the tree; further, the postcondition requires that the element be removed correctly and that the method returns `true` iff the element to remove (i.e., `i`) already exists in the input tree.

### 1.2.1 Generation at the Representation Level

How might we go about generating inputs to test the `remove` method? One way is to generate inputs at the representation level: create objects and set their field values. Consider, for example, manually generating the input tree `bst` that contains three elements (i.e., 1, 2, and 3) as shown in Figure 1-2. Figure 1-3 (a) illustrates this

```
package testera.examples;

class BinarySearchTree {
    Node root; // root node
    int size;  // number of nodes in the tree

    static class Node {
         Node left;
         Node right;
         int info;
   }

   // class invariant: ``this'' is a binary search tree: acyclicity;
   //                  elements in search order; size is number of nodes

   // precondition:
   // postcondition: element ``i'' is not in the tree in the post-state;
   //                all other elements that were in the tree in the
   //                pre-state are still in the tree in the post-state
   boolean remove(int i) { ... }
}
```

Figure 1-1: Java declaration of a class that implements binary search trees and a method that removes the given element from the input tree. Appendix A gives an implementation of `remove`.

```
BinarySearchTree bst = new BinarySearchTree();
bst.size = 3;
Node r = new Node();
bst.root = r;
Node l = new Node();
Node t = new Node();
r.left = l;
r.right = t;
r.info = 2;
l.info = 1;
t.info = 3;
```

Figure 1-2: Test generation at the representation level. Java code that generates a binary search tree by explicitly allocating objects and setting values of their fields.

input tree.

A key issue with manual generation at the representation level is that we need to manually establish structural invariants. This can be particularly cumbersome when augmenting an existing test input. Consider, for example, adding a new element (4,

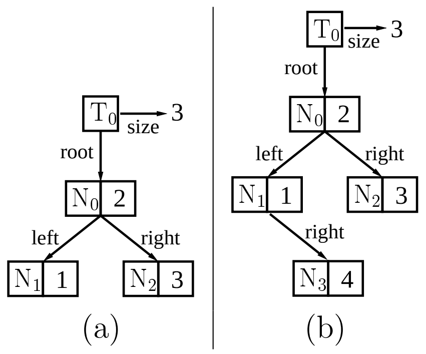

Figure 1-3: (a) Binary search tree with three nodes; (b) adding a new node that violates the search constraint. Each object has an identity, e.g., $N_0$ is the identity of the node object that represents the root node. Each node object contains an integer element that represents the value of field `info`. The fields `root`, `size`, `left` and `right` are appropriately labeled.

say) to `bst`:

```
Node lr = new Node();
l.right = lr;
lr.info = 4;
```

Figure 1-3 (b) displays the resulting tree. Since `l.info` = 1, `lr.info` must be larger than 1 (as otherwise we violate the search order). Setting `lr.info` to 4, however, still violates the search order since the node `lr` is in the sub-tree rooted at the left child of the root, which has a value (i.e., 2) less than 4. Moreover, we need to update the size value to 4, as now the tree has an additional node.

This example, despite its simplicity, illustrates the *global* nature of structural constraints that are hard to maintain locally. At first glance, it may seem that a sequence of random field value assignments is likely to generate a valid structure. But such an approach is infeasible, since the ratio of the number of valid structures to the number of candidate structures tends to zero as size is increased, and a random assignment is thus very likely to violate at least one of the constraints. As an example, consider arranging 3 nodes and 3 elements into a binary search tree using a random assignment of values to fields; there are $4 \cdot 4 \cdot (4 \cdot 4 \cdot 3)^3$ (or about 1.8 million)[1] possible assignments, but of these, only 21 represent valid binary search trees.

### 1.2.2 Generation at the Abstract Level

Input generation does not need to proceed at the representation level. We can generate inputs at an abstract level, by using an appropriate *construction sequence*. If

[1]For a candidate structure, the fields `root`, `left` and `right` can have one of four values (`null` or a reference to one of the three `Node` objects); the field `size` can have one of four values (0, 1, 2, or 3); and the field `info` can have one of three values (1, 2, or 3).

```
BinarySearchTree bst = new BinarySearchTree();
bst.add(1);
bst.add(2);
bst.add(3);
```

Figure 1-4: Test generation at abstract level. Java code that generates a binary search tree using a construction sequence.

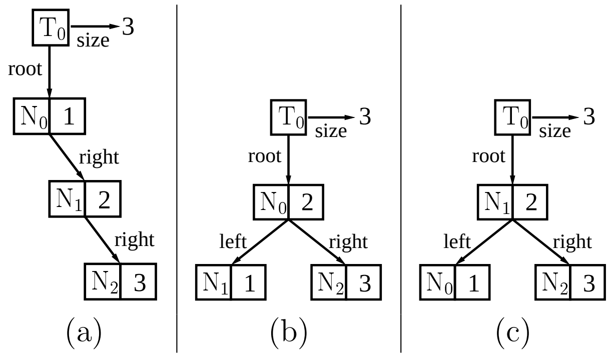

Figure 1-5: Binary search trees resulting from insertion of elements into an empty tree using the orders (a) `[1, 2, 3]`, (b) `[2, 1, 3]`, and (c) `[2, 3, 1]`. The trees (b) and (c) are isomorphic; they have the same branching structure and differ only in node identities.

we assume that we already have implemented a method, say `add`, to add new nodes to an existing tree, we can build an input tree using, for example, the construction sequence illustrated in Figure 1-4. It is now easy to add a fourth node: simply invoke `bst.add(4)`. The `add` method, assuming it is implemented correctly, automatically maintains the structural invariants for us.

Let us consider using this approach to generate a test suite. At a first glance, it may appear that it is sufficient to have one construction sequence for each tree size to build a reasonable test suite. Notice, however, that for the same set of elements, different orders of insertion can give rise to structurally different inputs. In our example, the order of insertion `[1, 2, 3]` (into an empty tree) gives rise to a list-like structure (where all nodes have their `left` fields set to `null`, Figure 1-5 (a)), whereas the order `[2, 1, 3]` gives rise to a balanced tree structure (Figure 1-5 (b)). To sufficiently test functionality of `remove` or another method that manipulates a binary tree structure, we need to generate several structurally different inputs.

So why don't we test on *all* sequences (up to a desired size)? Doing so typically produces many sequences that generate *isomorphic*[2] (or structurally equivalent) inputs. For example, the order of insertion `[2, 1, 3]` generates a tree that is isomorphic to the the tree that the order `[2, 3, 1]` generates (Figure 1-5 (c)). The problem worsens as the tree grows; for 10 nodes there are 10! or about $3.6 \times 10^6$ sequences,

[2]Section 6.2 formally defines isomorphism.

```
// binary tree
all n: this.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
}

// elements in correct order
all n: this.root.*(left+right) {
  all nl: n.left.*(left + right) | nl.info < n.info
  all nr: n.right.*(left + right) | nr.info > n.info
}

// size ok
this.size = #root.*(left + right)
```

Figure 1-6: Class invariant for `BinarySearchTree`.

whereas there are only 16,796 *nonisomorphic* (or structurally inequivalent) binary search trees [81]. Two inputs that are isomorphic elicit identical program behavior, for all programs. Therefore, there is no advantage in testing a program on more than one input from a set of isomorphic inputs.

### 1.2.3 Our Approach: Representation-level Generation from Constraints

The key idea in our approach is to generate structures at the representation level from their defining constraints given as formulas in first-order logic. We enumerate tests by solving the constraints using off-the-shelf SAT technology. To make systematic testing practical for real systems, we restrict test generation to nonisomorphic inputs.

The TestEra framework [66] presents an embodiment of these ideas for testing Java programs. TestEra generates tests using preconditions and checks correctness using postconditions as test oracles.

Our notation for expressing constraints allows succinct and declarative descriptions of a large class of graph-based structures [24] including both acyclic (e.g., red-black trees) and cyclic (e.g., doubly-linked circular lists) structures and also array-based structures such as hashtables or priority queues. In principle, we could express structural constraints of any data structure that a Java program implements.

We next illustrate a usage scenario of TestEra.

#### Writing Constraints

Recall the `BinarySearchTree` example. To generate inputs for the method `remove` the user provides the class invariant of `BinarySearchTree`. The invariant forms the precondition of `remove` and can be formulated in Alloy [43], a first-order logic based on sets and relations[3], as shown in Figure 1-6.

The invariant contains three formulas, implicitly conjoined. The first formula uses universal quantification (`all`) to state that `this` is a binary tree, in particular: (1) it is not possible to start traversal at any node in the tree and arrive at the same node, i.e., there are no directed cycles; (2) each node has at most one incoming edge labeled `left` or `right`; and (3) for each node, the `left` and `right` child cannot be the same node. The second formula states that the elements in the tree appear in the correct order, in particular, the element in a node is bigger (smaller) than all elements in the subtree rooted at its left (right) child. The third formula states that the value of the size field equals the number of nodes reachable from `root`.

The postcondition of `remove` includes the class invariant and additionally requires that the element be removed correctly and the method return the correct `boolean` value:

```
// element removed correctly
this.root.*(left + right).info = this.root`.*(left` + right`).info` - i

// result correct
@result = true <=> i in this.root`.*(left` + right`).info`
```

The backtick character '`' denotes field traversal in the pre-state. All other field traversals are in the default state, which is the pre-state for a pre-condition and the post-state for a post-condition. The keyword `@result` refers to the result of the method. This notation belongs to a veneer on Alloy that we have defined for specifying Java programs. We describe the details of our veneer in Chapter 4.

### Test Execution and Correctness Checking

Given the Java bytecode for `BinarySearchTree` and `BinarySearchTree$Node`, the class invariant and postcondition above, and a bound on the input size, TestEra generates test inputs and executes the method on each input to check the method's correctness.

As an illustration of TestEra's checking, consider erroneously replacing the condition `(info < current.info)` with `(info > current.info)` in the body of `remove` (Appendix A); this results in a failed search for input element unless the element belongs to the root node. Using the default bound[4] of three, TestEra detects violation of the correctness criterion.

A counterexample generated by TestEra consists of concrete Java input and output objects, which can be displayed in textual form using their `toString` methods or saved to disk using Java's support for serialization. Figure 1-7 graphically illustrates a TestEra counterexample that witnesses the error (we introduced) in `remove`. In this counterexample, the element to be deleted, with value `2`, still exists in the input tree

[3]For ease of exposition, we present a slight variant of Alloy. To compare integer values, we actually use library functions, e.g., instead of `a < b`, we write `LT(a, b)`.

[4]A bound of two is actually sufficient to reveal this bug.

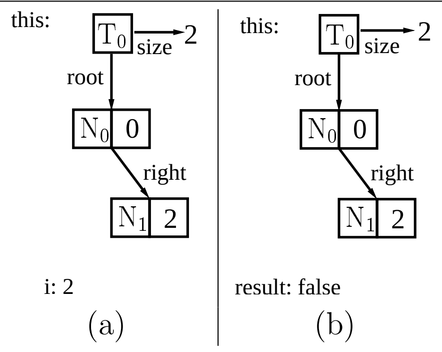

Figure 1-7: Counterexample to correctness of `remove`. Invoking `remove` on `this` and `i` in (a) the pre-state returns `false` and results in (b) the post-state.

in the post-state and the method returns `false` despite that the element exists in the input tree—a postcondition violation.

## 1.3 Contributions

This dissertation makes the following contributions:

- It describes a method for systematic constraint-based test generation for programs that take as inputs structurally complex data.

- For structure enumeration, where constraints are expressed in first-order logic, it describes how to manually construct first-order logic formulas to completely break symmetries, so that enumeration generates exactly one instance from each isomorphism class.

- It presents the design and (prototype) implementation of TestEra, a novel framework for automated specification-based testing of Java programs using SAT.

- It describes how to automatically translate data between an abstract domain and a concrete domain. Even though we use these translations for testing, they can be used in various other contexts, such as runtime monitoring and correcting of program behavior.

- It gives experimental evidence on the feasibility of using off-the-shelf SAT solvers for systematic generation of high quality test suites; the experiments witness efficient enumeration for complex data structures from the Java Collection Framework and identification of significant bugs in standalone applications.

- It describes a compelling application of SAT solvers that suggests that solution enumeration is an important feature that merits research in its own right. To the best of our knowledge, this is the first such application in software testing.

### 1.3.1 Test Generation as Constraint Solving

An input constraint defines desired inputs; for object-oriented programs these constraints often express structural properties.

Given the undecidable nature of these constraints, we bound the universe of discourse and find solutions that exist within the bound. Even though typically the bounded universe of possible solutions is still very large say for an exhaustive search to feasibly enumerate them, we  show that it is practical to enumerate solutions to such constraints using SAT technology: we translate constraints into propositional formulas, solve the formulas using off-the-shelf SAT solvers, reify solutions into concrete test inputs. The ability to efficiently enumerate allows us to overcome a key limitation of the current practice of testing: lack of a systematic way to generate a sufficiently large number of tests.

Using constraints to represent inputs is not a new idea and dates back at least three decades [19, 42, 56, 74]. Most of the prior work, however, has considered constraints on primitive data, such as integers and booleans, and not on structurally complex data.

Our approach generates tests from input constraints that specify the underlying data structures at the concrete level. This form of specification often appears in object-oriented software in terms of class invariants and method preconditions. Other common forms are algebraic [38] and model-based specifications [83], which typically introduce a level of abstraction. As we noted in Section 1.2.2, test generation for complex structures at an abstract level can result in a very high proportion of isomorphic structures.

### 1.3.2 Systematic Testing

Testing has traditionally been practiced in an ad hoc fashion. In cases where developers have a sufficiently large suite, they would run tests for as long as the time-to-market permits. These tests, however, are typically constructed by hand using the developers' intuition and do not necessarily test the functionality of the program to a desirable extent. We test the program *systematically*: the program is tested on all inputs up to a given bound[5].

Theoretically speaking, systematic testing guarantees program correctness only for all inputs that are within the given bound. In practice, however, systematic testing is likely to fare reasonably well in identifying subtle errors that are hard to detect otherwise. This is particularly likely to be the case for testing programs with structurally complex inputs. In such programs, the programmer typically implements code that handles structurally *different* cases, e.g., the `remove` method should behave correctly irrespective of whether a node in the input tree has zero, one, or two children.

[5]The notion of checking for errors systematically forms the foundation of the field of model checking [18]: a finite state model of a system is constructed and exhaustively checked for errors. Even though there is a lot of recent work in applying model checking techniques to software, the focus of this work has been on checking properties of event sequences in closed reactive systems and not on checking properties of data structures.

A suite consisting of all small inputs is likely to contain a representative of several of these cases. As a recent study indicates, for a variety of benchmark data structures, it is indeed possible to achieve complete statement and branch coverage by testing on all small inputs [65].

### 1.3.3 Inequivalent Inputs

The feasibility of systematic testing depends not only on the ability to generate inputs but also on the ability to test the program against the generated suite. As we have already illustrated, even for small input sizes, the set of all possible inputs may be very large. To make systematic testing practical, we restrict generation to inputs that are inequivalent (up to graph isomorphism) by conjoining, to the basic structural constraints, constraints that (1) define a mapping from a structure to its canonical form and (2) restrict generation to structures that map to themselves. Even though we require the user to provide the canonicalization constraints, we show how to write them using a simple idea of traversal. It is worth pointing out that our notion of equivalence among inputs is independent of any particular implementation: whatever program the inputs are passed to, the outcomes for equivalent inputs will be the same.

### 1.3.4 State Encoding

Mutation of state is a key aspect of object-oriented programs. Correctness criteria may relate states of a program at different control points during its execution. For example, method postconditions may relate values of an object's fields in the pre-state with the corresponding values in the post-state. To allow using SAT for checking rich behavioral properties, various encodings of mutable state of a Java program have been developed [53, 66, 89]. We present a novel encoding here. An advantage of our encoding is that it can be used in a modular fashion in a variety of domains, e.g., in modeling systems that have states with structure and transitions between states.

### 1.3.5 Experiments

We performed experiments to evaluate (1) the feasibility of using SAT solvers in solving complex structural constraints and generating nonisomorphic structures including some that are implemented in the Java Collection Framework [87] and (2) the use of TestEra in checking stand-alone applications. The initial experiments with TestEra exposed bugs in a naming architecture for dynamic networks [51] and a part of the Alloy 1.0 analyzer [66]; these bugs have now been corrected. More recently, we have applied TestEra to test a fault-tree solver [31] developed for NASA; TestEra exposed significant bugs in the fault-tree solver [86].

### 1.3.6 SAT Application

Our work presents a novel application of SAT solvers in software testing. The application requires a solver that can enumerate *all* solutions; this contrasts with previous SAT applications in the domains of AI planning [49], hardware verification [12], software design analysis [50], and code analysis [89]. These applications typically use a solver to find one solution, e.g., one plan that achieves a desired goal or one counterexample that violates a correctness property. Hence, most modern SAT solvers are optimized for finding one solution, or showing that no solution exists. That is also how SAT competitions [1] typically rank solvers.

Currently available versions of most modern SAT solvers, including zChaff [71], BerkMin [33], Limmat [11], and Jerusat [72], do not support solution enumeration at all[6], let alone optimize it. We hope that our application will motivate research in solution enumeration and will provide a new dimension at SAT competitions.

Our work provides a set of formulas that can be used to compare different solvers in their enumeration; these formulas fall into the (satisfiable) "industrial" benchmarks category for SAT competitions and are available online at:

`http://mulsaw.lcs.mit.edu/alloy/sat03/index.html`

We also present a performance comparison between mChaff and relsat in enumerating a variety of benchmark data structures.

## 1.4 Organization

This dissertation has three fundamental components:

- A basic description of our approach, including how to express constraints, how to solve constraints, and how to restrict test generation to inequivalent inputs (Chapters 3, 4, and 6).
- The embodiment of our approach into the TestEra framework for testing Java programs (Chapter 5).
- An assessment of our approach using a variety of case-studies and a comparison with related work (Chapters 7, 8 and 9).

In a little more detail these components are as follows.

Chapter 2 describes some relevant basic notions of software testing and our criterion for generating tests.

Chapter 3 gives the fundamentals of the Alloy specification language and its automatic tool support, and describes how we model a particular state (the heap) of a Java program in Alloy. Readers already familiar with Alloy may want to skip the sections on Alloy and the Alloy Analyzer.

[6]Support in zChaff is under development and available in an internal version. Support in BerkMin is planned for the next version. There is no plan to add support to Jerusat. (Personal communication with the authors of the solvers.)

Chapter 4 describes our veneer on Alloy, which enables building, in first-order logic, specifications of Java programs. We describe here how we model mutation of state in Alloy, and we present our specification notation. This chapter also describes some key aspects in which the semantic data model of Alloy differs from that of Java; the TestEra user needs to take these differences into account while writing specifications.

Chapter 5 presents the TestEra framework. We explain the key algorithms TestEra implements to generate Alloy specifications and data translations for test execution.

Chapter 6 describes how TestEra users can construct Alloy formulas that allow enumeration of exactly non-isomorphic instances.

Chapter 7 describes various case-studies that we have performed using our prototype implementation to evaluate the feasibility of using SAT for structure generation and TestEra for testing standalone applications.

Chapter 8 presents a discussion of the testing approach taken by TestEra. In particular, we point out both some salient features and inherent limitations. Also we describe here some future extensions and applications of the basic approach.

Chapter 9 presents an overview of the related work, in particular related work on specification-based testing, software model checking and static analysis.

Chapter 10 summarizes the dissertation and presents some final thoughts. This chapter is self-contained.

# Chapter 2

# Software Testing

The aim of testing a program is to find bugs. The process of testing consists of generating a set of test inputs, called a *test suite*, running the program against these inputs, and checking if each output is as expected. If an output is not as expected, the corresponding input witnesses an error and reveals a bug in the program or an error in the formulation of the correctness criterion.

This chapter describes some fundamental concepts in software testing, which are relevant to understanding our approach to test generation. We also explain the key properties that we desire of test suites.

## 2.1 Testing Using Preconditions and Postconditions

We aim to test the behavior of a given program against a correctness specification. The specification states the *precondition*, which defines the properties that the program requires of its inputs, and a *postcondition*, which describes expected outputs. A precondition is a formula that can be evaluated on a given input. Inputs for which the formula evaluates to true represent inputs that the program allows. A postcondition is a formula that can be evaluated on a given input/output pair.

In object-oriented programs, *class invariants* define properties that objects of a class must satisfy. Methods that classes implement require (as their preconditions) the inputs to satisfy their respective class invariants and may require additional properties that relate different inputs with each other. The method postconditions can be viewed as test oracles that relate pre-states (i.e., the program state immediately before method invocation) and post-states (i.e., the program state immediately after method invocation): the postcondition is a formula that can be evaluated on a given pre-state and corresponding post-state.

It is worth emphasizing that a precondition is a property a method expects to be true (and therefore need not check) of its inputs at the point of invocation. And correct behavior (as specified by the postcondition) is only guaranteed if the invocation is on inputs that satisfy the precondition.

### 2.1.1 Need Relevant Tests

For programs that only use elementary data-types, generating tests is simple. For example, if a program considers as valid inputs any integer in a given range, it is easy to sample a test suite. However, for programs that take as inputs structurally complex data, generating even a few tests is non-trivial. Tests generated in a random fashion are likely to violate some structural constraint and therefore be irrelevant since such programs explicitly require inputs to satisfy all desired constraints.

Indeed, it may be desirable to observe the behavior of a program on inputs that violate the precondition. Relevant tests in such a scenario are with respect to the properties that the programmer desires the inputs to satisfy or violate.

### 2.1.2 What Tests to Use?

Given the ability to generate relevant tests, we need to determine what tests to use. Non-trivial programs effectively have an infinite number of possible inputs. Testing exhaustively on all inputs is not feasible. In practice, a program is typically tested for as long as the available resources allow.

Recall that we view a test suite as a set of inputs. The time required to test a program is determined by the size of the suite: the larger the suite, the longer it takes for testing to complete. Thus, deciding what inputs to actually use is a key decision to make when testing a program.

Suites can alternatively be viewed as sequences of inputs where the program is first tested against inputs that are more likely to find bugs. In this case, deciding how to order tests is a key decision to make. We do not consider ordering inputs here.

### 2.1.3 Test Adequacy

A *test adequacy criterion*—a set of conditions that a desired suite must fulfill—is defined to determine whether a generated suite tests a program to a desirable extent. Criteria can be defined both for *white-box* testing, where program implementation guides test generation, and for *black-box* testing, where specifications alone are used for selecting tests (and the program implementation is ignored) [10].

Common forms of criteria for white-box testing are to require that tests in a suite cover some aspect of implementation code, i.e., exercise all statements, branches or even paths (up to a certain length) in the program. A key advantage of white-box testing is that by taking into account the details of the implementation, it allows generating suites that are tuned to test a particular implementation. For example, consider a typical specification of a program for sorting a list of integers. The actual implementation might, for performance reasons, implement a combination of algorithms, say bubble-sort for input lists with less than 5 elements and quicksort for input lists with 6 or more elements. A black-box based approach, oblivious of the actual implementation, might miss testing an input with 6 or more elements and therefore fail to test the quicksort implementation. In this case statement coverage, would suffice to rule out such a suite.

In black-box testing, the structure of specification code is used to define the criterion [15]. For example, for a specification of a square root program that specifies how the program behaves if the input is less than, equal to, or greater than 0, we could use the criterion that the suite should consist of some integer less than 0, the integer 0, and some integer greater than 0. A key advantage of black-box testing is that it catches errors of omission, or cases that the implementation does not handle (and therefore fails on). For example, consider a program that is intended to compute square root (to the nearest integer) but erroneously returns 0 on all inputs:

```
int squareRoot(int i) {
    return 0;
}
```

A white-box based approach that aims to test all paths in the program may test the program against the suite containing only the input zero and incorrectly deem the program correct. Furthermore, the cost of selecting a test in black-box testing is independent of the size of the implementation, which is typically (much) longer than the specification.

Adequacy criteria may be based on both the specification and the implementation. For example, a criterion could require covering all branches in the implementation and all conditions in the specification.

The focus of our work is black-box testing, where specifications take the form of input constraints and test oracles. Accordingly, we restrict our discussion to this case.

### 2.1.4 Redundancy in Test Suites

A test suite may contain redundancy in the following sense: it contains more than one test that witnesses a single bug. Much as we would like to minimize the cost of test execution by generating suites that do not have redundancy, for a given program and correctness property, it is not possible, in general, to eliminate all such redundancy—doing so would amount to determining program correctness [92].

Nonetheless, we can eliminate some redundant tests from the suite by disallowing inclusion of *equivalent* tests. We consider two tests equivalent if they induce identical observable behaviors on any (deterministic) program. For programs that operate only on elementary data-types, only identical inputs are equivalent. For more general programs, inputs that are not identical may still be equivalent. Pointer-based structures, such as a linked list or a binary search tree are examples of such inputs.

Testing on two (or more) inputs that are equivalent is a waste of resources: if the program behaves correctly (respectively incorrectly) on one, it behaves correctly (respectively incorrectly) on all that are equivalent to the one. The notion of equivalence is presented in Weyuker and Ostrand's work [92].

### 2.1.5 Dense Suites

For any two inputs that are not equivalent, it is possible to write a program that exhibits different behaviors on them. Therefore, by leaving out an input that is not equivalent to others already in use, we risk missing out on detecting a buggy behavior.

Since the set of inequivalent inputs for (non-trivial) programs is effectively infinite, exhaustive testing of a program is not possible even if test generation is restricted to inequivalent inputs. However, it is feasible to test programs against *dense* suites, i.e., suites that contain all inequivalent inputs within certain bounds on the input size. For inputs that can be represented using graph-based structures, we use the number of nodes in the graph and the number of elements in relevant elementary domains as a measure for the input size. As an example, a dense suite for testing a method that removes the root node of a binary tree and arranges any remaining nodes into a tree may contain all inequivalent trees with up to 5 nodes.

Of course, testing against dense suites checks program correctness only for those inputs bounded by the given size. In practice however, such an approach is likely to fare well in identifying subtle errors that are hard to detect otherwise (using the traditional approach of manual test generation). This is particularly likely to be the case for testing programs with structurally complex inputs. In such programs, the programmer typically implements code that handles structurally *different* cases, e.g., the `remove` method should behave correctly irrespective of whether a node in the input tree has zero, one, or two children. A suite consisting of all small inputs is likely to contain a representative of several of these cases.

In contrast, testing against a randomly generated suite is not as likely to fare well. First, for complex structures the ratio of the number of valid structures to the number of candidate structures tends to zero as structure size increases. This makes it hard to generate such data in a random fashion and we do not know how to do so. Second, even if we assume that we have a stream of relevant tests available and we can sample them randomly, the likelihood that a random sample would cover all structurally different cases that the program handles is low.

## 2.2 Our Criterion

We test a program against dense suites for as long as time permits: our test adequacy criterion is to test on all inequivalent inputs systematically (starting from the smallest ones and iteratively increasing the input size) for as long as the available time permits. One key difference between our approach and traditional manual approaches is the sheer volume of tests: we typically test a program on several hundred thousand (structurally complex) inputs; doing so using manual generation would simply be infeasible.

A question that naturally arises is how effective this criterion is in increasing the reliability of software. The experimental results indicate that suites that meet this criterion are able to find subtle bugs that went unnoticed despite years of use (Chapter 7). Also, a recent study indicates that it is feasible to generate suites based on this criterion and achieve full statement and branch coverage in a variety of benchmark data structure implementations [65].

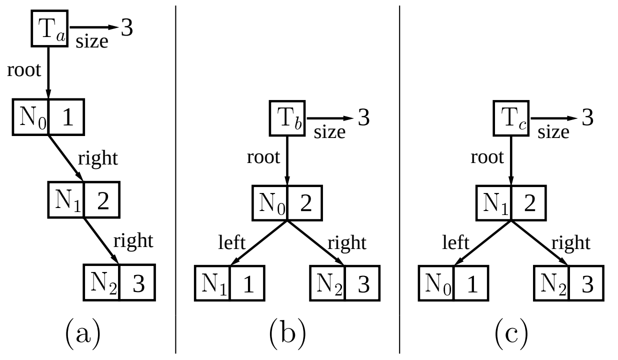

Figure 2-1: Binary search trees resulting from insertion of elements into an empty tree using the orders (a) `[1, 2, 3]`, (b) `[2, 1, 3]`, and (c) `[2, 3, 1]`. The trees (b) and (c) are isomorphic; they have the same branching structure and differ only in node identities.

### 2.2.1 Equivalent Tests for Object-Oriented Programs

For object-oriented programs, we use the notion of *structural isomorphism* (which we define formally in Chapter 6) to define equivalence among inputs. We assume that object identities are used for comparisons only. In Java, it is possible to use `System.identityHashCode(...)` to do operations with object identities besides comparison. However, such programs usually indicate bad style and are rare. They can be identified using a simple static analysis and we do not consider them here.

As an illustration of equivalent inputs, consider `BinarySearchTree`'s `remove` method (shown in Appendix A, which takes two inputs: a tree and an integer. The following set consists of two inputs that are equivalent: $\{(T_b,\ i),\ (T_c,\ i)\}$, where $T_b$ is the balanced tree in Figure 2-1(b), $T_c$ is the balanced tree in Figure 2-1(c) and $i$ is an integer, since $T_b$ and $T_c$ are structurally isomorphic as witnessed by the permutation of nodes $(N_0\ N_1)$.

The set $\{(T_a,\ i),\ (T_c,\ i)\}$, where $T_c$ and $i$ are as before but $T_a$ is the tree in Figure 2-1(a), does *not* consist of equivalent inputs; this is because it is possible to implement `remove` in a way that it performs correctly on the input $(T_a,\ i)$ but incorrectly on the input $(T_c,\ i)$. To see this, consider as a witness the implementation that first checks if the input tree is not balanced and if so (erroneously) raises an exception but otherwise correctly removes the given element. Similarly the set $\{(T_a,\ i),\ (T_a,\ j)\}$, where $T_a$ and $i$ are as before but integer $j \neq i$, does not consist of equivalent inputs.

It is worth emphasizing that testing only on nonisomorphic inputs does not affect the ability to detect bugs. Moreover, in object-oriented (such as Java) programs re-running the same test (say for regression testing) does *not* necessarily test the program on an input identical (with respect to identities of constituent objects) to the one used before. For example consider a scenario for regression testing where we save a concrete test to the disk using serialization [87] of its constituent objects. Loading the test from the disk using de-serialization creates new objects, which generates a test isomorphic (and not necessarily identical) to the one that was serialized. Similarly, if the test is saved to disk in an abstract form, such as a construction sequence, different invocations of the same construction sequence generate distinct but isomorphic tests.

Furthermore, in generating inequivalent inputs, it is essential to take into account isomorphism at the concrete representation level. Even though in certain cases it may be possible to directly generate construction sequences that produce a set of nonisomorphic structures, a priori determining such sequences requires program analysis of the implementations of the methods and is not tractable in general.

In addition to isomorphism among inputs, we can exploit the developer's intuition in defining equivalent inputs (and further reducing the size of generated suites). In particular, if the developer believes that program behavior (with respect to the correctness property being checked) is identical on a certain class of inputs, we can add, to the original input constraints, constraints that restrict generation to not more than one input from that class.

# Chapter 3

# Modeling structures

Structurally complex data abound in modern software. The key issue in automating testing is automating the generation of input data. Two issues that make generating structures hard are that they involve references and sharing, and that only a small proportion of arbitrarily selected structures are well-formed.

In this chapter, we first give some examples of how structures arise. Next, we describe how we model states of an object-oriented program in a first-order setting. We intuitively describe the basics of Alloy [43], a first-order declarative language based on relations, as we introduce it. We illustrate how to model the structural constraints of a variety of benchmark structures. We also explain relevant details of Alloy and the Alloy Analyzer [46] at the end of this chapter.

Given user-provided precondition and postcondition constraints, TestEra, our tool for testing Java programs, builds Alloy specifications, which include both the constraints for test generation and correctness checking, and also models of the classes that are used in testing. This enables using the Alloy Analyzer for solving the constraints. We describe our notation for expressing preconditions and postconditions in Chapter 4 and how TestEra builds Alloy specifications and translates data between Alloy and Java domains in Chapter 5.

## 3.1 Structures are Pervasive

Think of testing a compiler. It requires generating input programs, which are structures that are constrained by the semantic and syntactic constraints of the underlying programming language. Or consider a file system, whose structural integrity is constrained by various properties, such as that the root directory has no parent and that no directory is an ancestor of itself.

Structural complexity may arise from performance concerns. For example, balanced binary trees [24] provide efficient insertion and retrieval of (ordered) data. Red-black trees are a common form of balanced binary search trees and have defining structural constraints, such as that red nodes have black children and that the number of black nodes along any path from root to leaf is the same.

Hierarchy also gives rise to structures. Intentional naming [3] allows services to be accessed using a description of their properties, without a priori requiring knowledge of their locations. Fault trees [31] use a hierarchical arrangement to represent the overall failure of a system as a combination (using boolean gates) of failures of basic components of the system.

In Java programs, structures consist of objects that are dynamically allocated on the heap. Testing Java programs, therefore, has to take into account the heap. We describe next our basic model of the heap.

## 3.2 Modeling the Heap

To model the heap of a Java program in Alloy, we take a relational view: we view the heap of a Java[1] program as a labeled graph whose nodes represent objects and whose edges represent fields. The presence of an edge labeled `f` from node `o` to `v` says that the `f` field of the object `o` points to the object `v` or has the primitive value `v`. Mathematically, we treat this graph as a set (the set of nodes) and a collection of relations, one for each field. We partition the set of nodes according to the declared classes[2] and partition the set of edges according to the declared fields. We model primitive values also as nodes but define appropriate Alloy functions to support common operations on them.

A particular program state is represented by an assignment of values to these sets and relations. Since we model the heap at the concrete level, there is a straightforward isomorphism between program states and assignments of values to the underlying sets and relations. This isomorphism allows natural translations between data in the abstract domain and the concrete domain, which are required for test generation and correctness checking.

A key advantage of our model of the heap is that it is in accordance with the intuition programmers typically have. Furthermore, our notation for writing constraints allows path expressions, which represent heap traversals in an intuitive fashion. Also, the relational view enables us to use the automatic analysis support of the Alloy Analyzer.

### 3.2.1 Examples

#### Singly-linked list

Consider, as an example, the following class declaration of a singly-linked list of integers:

```
class List {
    Entry header; // first node
```

[1]For languages, such as C and C++, that allow pointer arithmetic and arbitrary conversions between integer values and memory addresses, we would need a different view. For type-safe subsets of such languages, we can still use the relational view.

[2]In this work, we do not address subclassing; it is discussed elsewhere [53].

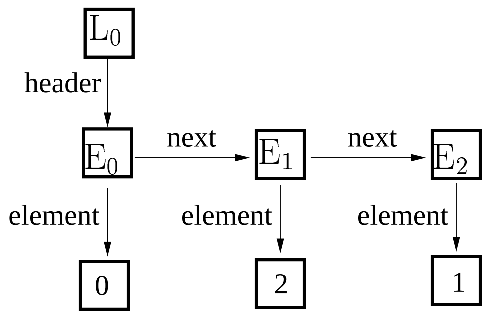

Figure 3-1: Singly-linked list with three elements shown as a graph. Each node (shown as a box) is labeled with either an object identity or a primitive value that the node represents. The edges are labeled according to the fields they represent.

```
      static class Entry {
           Entry next;
           int element;
     }
  }
```

The list consists of `Entry` objects that represent the nodes of the list. The `header` fields represents the (possibly `null`) first[3] node in a list. The basic model of heap for this example consists of three sets, each corresponding to a class or a primitive type[4] declared above:

```
List
Entry
Int
```

and three relations, each corresponding to a field declared above:

```
header: List -> Entry
next: Entry -> Entry
element: Entry -> Int
```

Let's assume the class invariant of `List` requires lists to be acyclic. The following Alloy formula expresses this:

```
all e: this.header.*next | e !in e.^next
```

The path expression `this.header.*next` uses the reflexive transitive closure operator ('*') to denote the set of all nodes reachable from the `header` node of the list referenced by `this`. The operators '^' and '!' denote respectively transitive closure and logical negation. The first formula uses universal quantification (`all`) to state that `this` is

[3]Some common implementations of linked lists (e.g., as in Java Collection Framework) treat the `header` node as a sentinel node. We take a simpler view here.

[4]We model primitive integers using the set `Int`, which is provided in the Alloy library.

an acyclic list: it is not possible to start traversal at any node in the list and arrive at the same node, i.e., there are no directed cycles.

Figure 3-1 illustrates a list that represents the following valuation of the sets and relations that we use in our model:

```
List = { L0 }
Entry = { E0, E1, E2 }
Int = { 0, 1, 2 }

header = { (L0, E0) }
element = { (E0, 0), (E1, 2), (E2, 1) }
next = { (E0, E1), (E1, E2) }
```

### Binary search tree

As another example, consider the class declaration of `BinarySearchTree` (introduced in Section 1.2):

```
class BinarySearchTree {
    Node root; // root node
    int size;  // number of nodes in the tree

    static class Node {
         Node left;
         Node right;
         int info;
   }
}
```

The basic model of heap for this example consists of three sets, each corresponding to a class or a primitive type declared above:

```
BinarySearchTree
Node
Int
```

and five relations, each corresponding to a field declared above:

```
root: BinarySearchTree -> Node
size: BinarySearchTree -> Int
info: Node -> Int
left: Node -> Node
right: Node -> Node
```

Recall the class invariant for binary search trees: acyclicity, orderedness of elements, and representation of the number of nodes by the `size` field. This can be represented in Alloy as:

```
// binary tree
all n: this.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
}

// elements in correct order
all n: this.root.*(left+right) {
  all nl: n.left.*(left + right) | nl.info < n.info
  all nr: n.right.*(left + right) | nr.info > n.info
}

// size ok
this.size = #this.root.*(left + right)
```

The invariant contains three formulas, implicitly conjoined. The path expression `this.root.*(left + right)` denotes the set of all nodes reachable from `root` of the tree referenced by `this`. The operator '#' denotes set cardinality. The operator '~' denotes backward traversal. In Alloy, the formula `no e` is true when `e` denotes a relation containing no tuples. Similarly, `some e`, `sole e`, and `one e` are true when `e` has some, at most one, and exactly one tuple respectively.

The first formula above uses universal quantification (`all`) to state that `this` is a binary tree, in particular: (1) it is not possible to start traversal at any node in the tree and arrive at the same node, i.e., there are no directed cycles; (2) each node has at most one incoming edge labeled `left` or `right`; and (3) for each node, the `left` and `right` child cannot be the same node. The second formula states that the elements in the tree appear in the correct order, in particular, the element in a node is bigger (smaller) than all elements in the subtree rooted at its left (right) child. The third formula states that the value of the size field equals the number of nodes reachable from `root`.

### 3.2.2 Handling Null

We model the value `null` as the empty set. Fields of a reference type thereby become partial functions. In particular, to say the value of field `f` in object `o` is non-`null`, we use the formula `some o.f`; similarly for `null`, we write `no o.f`.

This approach is not sufficient in general. For example, it does not allow correctly expressing in a direct fashion the formula `null in S`, for some set `S`. In many cases, however, specifications for Java programs that talk about membership of `null` in a set can still be formulated, often at the expense of representation exposure and more complex formulas. As an example, suppose that we have a red-black tree [24] implemented as a Java `TreeMap`. Consider expressing the formula that `null` belongs to the set of all values in a `TreeMap t`. We cannot write the formula as `null in values(t)`, where `values` is a function that returns the set of values in the given map. However we can instead write the formula as `some n in t.root.*(left + right) | no n.value`, which says that there exists a node in the tree that has the value `null`.

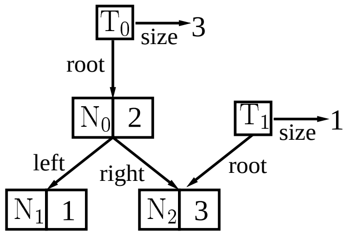

Figure 3-2: Heap snapshot that shows two trees $T_0$ and $T_1$ that share a node. The values of field `info` are shown inside the nodes. All other fields are labeled appropriately.

A more general approach is to model `null` explicitly [53]. We can model `null` as the only atom that is common to all distinct classes; class hierarchies can be modeled using subsignatures.

### 3.2.3 Aliasing

Our model accounts naturally for aliasing (or sharing) of objects on the heap. Consider, for example, the particular heap snapshot illustrated in Figure 3-2, which show two trees that share a node. According to our model they are represented by the following valuation of sets and relations:

```
  BinarySearchTree = { T0, T1 }
  Node = { N0, N1,  N2 }
  Int = { 1, 2, 3 }

  root = { (T0, N0), (T1, N2) }
  size = { (T0, 3), (T1, 1) }
  info = { (N0, 2), (N1, 1), (N2, 3) }
  left = { (N0, N1) }
  right = { (N0, N2) }
```

#### Cyclic Structure Example: Doubly-linked list

Structures that have cycles can, of course, be captured using our model. As an illustration consider doubly-linked circular lists with sentinel header nodes, as implemented by `java.util.LinkedList`. In a sentinel node, the `element` field always has the value `null`, the `next` field points to the first element of the list and the `previous` points to the last element of the list; for an empty list, the `next` and `previous` fields of the `header` node point to the sentinel node itself. The following Java code gives the class declaration:

```
class LinkedList {
    Entry header;
    int size;
```

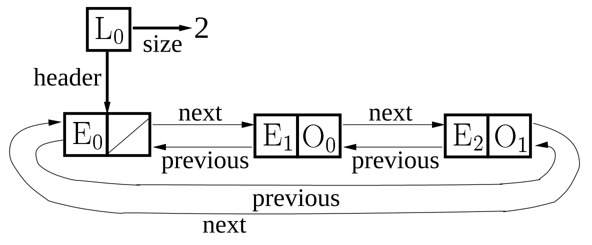

Figure 3-3: Doubly-linked circular list with size 2. List $L_0$ has sentinel header node $E_0$ whose element is the value `null`.

```
    static class Entry {
        Object element;
        Entry next;
        Entry previous;
    }
}
```

The class invariant can be expressed as follows.

```
  // size ok
  this.size =  #(this.header.*next - this.header)

  // header is sentinel
  some this.header && no this.header.element

  // next is transpose of previous
  all e1, e2: this.header.*next |
    e1 in e2.next <=> e2 in e1.previous

  // list is circular
  all e: this.header.*next | some e.next && some e.previous
```

The invariant contains four formulas, implicitly conjoined. The set operator '-' denotes set difference. Alloy provides the standard logical operators: `&&` (conjunction), `||` (disjunction), `=>` (implication), `<=>` (bi-implication), and `!` (negation).

The constraint above requires that the `size` of the list is the number of nodes other than the sentinel node; `header` is the sentinel node of the list and contains no `element`; values of `next` and `previous` are consistent; and list is circular or simply every node has non-`null` values of the fields `next` and `previous`. Figure 3-3 illustrates a list of size 2.

### 3.2.4 Handling Arrays

We model single-dimensional arrays as follows. The array-type `T[]`, where `T` is not an array-type is modeled by the set `T'`, where each atom of `T'` represents an array object, and the relations

```
length: T’ -> Int
contents: T’ -> Int -> T
```

A tuple $(a, l)$ in relation `length` represents that array $a$ has length $l$. A tuple $(a, i, t)$ in relation `contents` represents that array $a$ has element $t$ at index $i$. The relations are constrained to represent valid arrays: (1) each array has a length, (2) an array contains one element (which may be the value `null`) at any (valid) index, and (3) arrays can only indexed using (positive) integers that are less than the length.

If a field `f` in class `C` is of a single-dimensional array-type `T[]`, we model `f` as the relation `f: C -> T’`. We describe the model for multi-dimensional arrays once we introduce more details of Alloy (in Section 3.3.3).

A consequence of our model of `null` and arrays is that any expression that dereferences `null` or accesses an array index that is out of array bounds evaluates to the empty set and not a Java exception such as `NullPointerException` or `ArrayIndexOutOfBoundsException`. We do not present a treatment of exceptions in this work.

To avoid building erroneous specifications, TestEra users need to take the model of `null` and arrays into account. For example, the formula `v.f.g = w.h[i]` (for quantified variables `v` and `w`, fields `f`, `g` and `h`, and integer `i`) evaluates to true if `v.f` is `null` and index `i` is out of bounds of array `w.h`, whereas according to Java semantics the left-hand-side and the right-hand-side of the above equality do not represent the same value. In other words, TestEra users may need to explicitly check for relevant exceptions in building their specifications (as the example that follows illustrates). However, in common cases that use path expressions to build sets of reachable objects, these checks are not necessary.

### Array-based Structure Example: Heap-array

We next illustrate specifying an array-based structure, using the *heap* (or priority queue) data structure [24].

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 $n$. The following Java code declares an array-based heap:

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

The class invariant can be defined as follows[5].

```
  // array is not null
  some this.array
```

[5]As pointed out before, the actual constraints on integers we write for test generation use Alloy functions for arithmetic operations. For ease of exposition, we use common arithmetic notation.

```
// size is within array bounds
this.size >= 0
this.size <= this.array.length

// the array elements that are in the heap are not null
// and follow the right ordering
all i: Int |
  i >= 0 && i < this.size =>
    some this.array[i] && (i > 0 => this.array[i] <= this.array[(i-1)/2])

// the array elements that are not in the heap are null
all i: Int | i >= this.size && i < this.array.length => no this.array[i]
```

We next explain relevant details of the Alloy specification language and point out how, in an Alloy specification, constraints that define structures fit together with declarations of sets and relations that model the heap. We also describe relevant details of the Alloy Analyzer.

## 3.3 Alloy

Alloy is a declarative specification language based on relations. An Alloy specification is a sequence of paragraphs that can be of two kinds: signatures, used for construction of new types and declaration of sets and relations, and a variety of formula paragraphs, used to record constraints. Each specification starts with a `module` declaration that names the specification; existing specifications may be imported into the current one using `open` declarations. Figure 3-4 presents a specification that brings together our model of binary search tree. We use this example to explain the basics of Alloy.

### 3.3.1 Signature Paragraphs

A signature paragraph introduces a basic (or uninterpreted) type, and a collection of relations (called *fields*) and constraints on field values. For example,

```
sig BinarySearchTree {
  root: option Node,
  size: Int
}

sig Node {
  left, right: option Node,
  info: Int
}
```

introduces `BinarySearchTree` and `Node` as sets of atoms. `BinarySearchTree` has two fields: `root` and `size`. `Node` has three fields: `left`, `right` and `info`. The field `root`

```
module bst

sig BinarySearchTree {
  root: option Node,
  size: Int
}

sig Node {
  left: option Node,
  right: option Node,
  info: Int
}

fun BinarySearchTree::ClassInvariant() { with this {
  // binary 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
  }

  // elements in correct order
  all n: root.*(left+right) {
    all nl: n.left.*(left + right) | nl.info < n.info
    all nr: n.right.*(left + right) | nr.info > n.info
  }

  // size ok
  size = #root.*(left + right)
} }

fun Main() {
  all b: BinarySearchTree | b..ClassInvariant()
}

run Main for 3 but 1 BinarySearchTree, 4 Int
```

Figure 3-4: Alloy specification for a binary search tree.

introduces a relation of type `BinarySearchTree -> Node`. The keyword `option` indicates that for each `BinarySearchTree` atom $b$ there is *at most one* `Node` atom $n$ such that the tuple $(b, n)$ belongs to the relation `root`; the absence of a keyword indicates *exactly one*.

The field `size` introduces a relation that has the type `BinarySearchTree -> Int`. Similarly, the fields `left` and `right` each introduce a relation of type `Node -> Node`, and the field `info` introduces a relation of type `Node -> Int`. `Int` is a library signature that models integers.

Fields of signatures may represent relations of arbitrary arity. For ternary relations, field declarations may include the cardinality markings '`?`' and '`!`'. The declaration

```
sig S {
  f: A ->? B
}
```

indicates that for each `S` atom $s$ and each `A` atom $a$, there is *at most one* `B` atom $b$ such that the tuple $(s, a, b)$ belongs to the relation `f`; the cardinality markings '`!`' likewise indicates *exactly one*.

A signature declaration may introduce a subset using the keyword `extends`:

```
sig T extends S {} // T is a subset of S
```

A singleton set may be declared using the keyword `static`:

```
static sig U {} // U is a (basic) set with cardinality one
```

Scalars are modeled as singleton sets.

## 3.3.2 Formula Paragraphs

We next describe Alloy expressions, operators and quantifiers that turn expressions into formulas, and the different kinds of formula paragraphs.

### Relational Expressions

The value of every expression in Alloy is a relation—that is a set of tuples of atoms. A relation may have any arity greater than zero and is typed. Sets are represented by unary relations and scalars as singleton sets. Scalars are viewed as singleton sets. Therefore, no distinction is made between `a`, {`a`}, {`(a)`}, where `a` is an atom, and all are represented as {`(a)`}.

Relations can be combined with a variety of operators to form expressions. The standard set operators—union '`+`', intersection '`&`', and difference '`-`'—combine two relations of the same type, viewed as sets of tuples.

The dot operator '`.`' is relational composition. For relations `p` and `q` where `p` has type $\texttt{T}_1$ `-> ... ->` $\texttt{T}_\texttt{m}$ and `q` has type $\texttt{U}_1$ `-> ... ->` $\texttt{U}_\texttt{n}$, if:

1. $\texttt{T}_\texttt{m}$ `=` $\texttt{U}_1$ and

2. `m + n > 2`

then `p.q` is a relation of type $\texttt{T}_1$ `-> ... ->` $\texttt{T}_{\texttt{m-1}}$ `->` $\texttt{U}_2$ `-> ... ->` $\texttt{U}_\texttt{n}$ such that for each tuple $(t_1, ..., t_m)$ in `p` and each tuple $(u_1, ..., u_n)$ in `q` where $t_m = u_1$, the tuple $(t_1, ..., t_{m-1}, u_2, ..., u_n)$ is in `p.q`. When `p` is a unary relation (i.e., a set) and `q` is a binary relation, `p.q` represents the standard image of `p` under `q`.

The unary operators '`~`' (transpose), '`^`' (transitive closure), '`*`' (reflexive transitive closure) and '`^`' (transitive closure) have their standard interpretation and can only be applied to binary relations.

For example, the path expression `root.*(left + right)` uses the reflexive transitive closure operator ('*') to denote the set of all nodes reachable from the `root` node.

The unary operator '`#`' denotes set cardinality.

## Formulas and Declarations

Expression quantifiers turn an expression into a formula. The formula `no e` is true when `e` denotes a relation containing no tuples. Similarly, `some e`, `sole e`, and `one e` are true when `e` has some, at most one, and exactly one tuple respectively. Formulas can also be made with relational comparison operators: subset (written '`:`' or `in`), equality ('=') and their negations (`!:`, `!in`, `!=`). So `e1: e2` is true when every tuple in (the relation denoted by the expression) `e1` is also a tuple of `e2`. Alloy provides the standard logical operators: `&&` (conjunction), `||` (disjunction), `=>` (implication), `<=>` (bi-implication), and `!` (negation); a sequence of formulas within curly braces is implicitly conjoined.

A *declaration* is a formula `v:e` consisting of a variable `v` and an arbitrary expression `e`. Quantified formulas consist of a quantifier, a comma separated list of declarations, and a formula. In addition to the universal and existential quantifiers `all` and `some`, there is `sole` (at most one) and `one` (exactly one).

## Functions, Facts and Assertions

A function (`fun`) is a parameterized formula that can be "invoked" elsewhere. For example, the function `f` declared as:

```
fun f(p1: T1, ..., pn: Tn) { ... }
```

has `n` parameters: `p1, ..., pn` of types `T1, ..., Tn` respectively. A function (with at least one parameter) may equivalently be declared as:

```
fun T1::f(p2: T2, ..., pn: Tn) { ... }
```

and its first parameter can then be referred to using the keyword `this` within the function body. The function `f` may be invoked as `f(a1, ..., an)` or alternatively using the receiver syntax as `a1..f(a2, ..., an)`. In the function body, the return value is referred to using the keyword `result`.

A `fact` is a formula that takes no arguments and need not be invoked explicitly; it is always true. An assertion (`assert`) is a formula whose validity is checked, assuming the facts in the model.

### 3.3.3 Example: Modeling Multi-dimensional Arrays

We next re-visit our model of single-dimensional arrays (Section 3.2.4) and describe how we model multi-dimensional arrays. The single-dimensional array-type `T[]` is modeled[6] by the signature:

```
sig T' {
  length: Int,
  contents: Int ->? T
}
{
  all i: Int | i >= length => no i.contents
}
```

Each atom of `T'` models an array object, which has a length and contains elements. The elements of an array object are modeled by the relation `contents`. For object `o` of class `C` that declares a field `f` of type `T[]`, the `i`th element of array `o.f`, written `o.f[i]` in Java, is modeled by `i.(o.f.contents)`.

Multi-dimensional arrays are modeled in a similar way[7]. We model the $k$-dimensional array-type `T[]`$\cdots$`[]`, where `T` is not an array-type, by the (new) signature `T'`$^k$ in a fashion similar to our model of `T[]` above except that[8]:

- the last column of the `contents` relation now is `T'`$^{(k-1)}$, which models the $k-1$-dimensional array-type `T[]`$\cdots$`[]`; and

- recursively a signature corresponding to each of the following array-types is also declared: $k-1$-dimensional `T[]`$\cdots$`[]`, ...,1-dimensional `T[]`.

For example, the following Alloy specification models a two-dimensional array type `T[][]`:

```
sig T'' {
  length: Int,
  contents: Int ->? T'
}
{
  all i: Int | i >= length => no i.contents
}
```

where `T'` is as defined above.

We next describe the Alloy Analyzer that provides the basic technology we use for solving structural constraints.

[6]We could alternatively model arrays using Alloy's library support for modeling sequences

[7]Our current prototype does not handle multi-dimensional arrays.

[8]The notation `T'`$^k$ stands for `T` followed by $k$ primes `''`.

### 3.3.4 Alloy Analyzer

The Alloy Analyzer [46] is an automatic tool for analyzing formulas in Alloy specifications. Since first order logic is undecidable, the analyzer limits its analysis to a finite universe of discourse. Given a formula and a *scope*—a set of numeric bounds that define the number of atoms for each basic type—the analyzer determines whether there exists a *model* of the formula (that is, an assignment of values to the sets and relations that makes the formula true) that uses no more atoms than the scope permits, and if so, returns it. The analysis [46] is based on a translation to a boolean satisfaction problem, and gains its power by exploiting state-of-the-art SAT solvers. The analyzer provides two kinds of analysis: *simulation* in which the consistency of a fact or function is demonstrated by generating a model of the constraints that the function represents, and *checking*, in which a consequence of the specification is tested by attempting to generate a model that represents a counterexample.

The models of formulas are termed *instances*. Consider our specification for `BinarySearchTree` (Figure 3-4). The function `Main` invokes the function `ClassInvariant` to constrain all trees to satisfy the binary search tree properties. The command:

```
  run Main for 3 but 1 BinarySearchTree, 4 Int
```

instructs the analyzer to generate valuations of sets and relations that satisfy the constraints enforced by the function `Main` and are within a scope of 3 for each basic set except `BinarySearchTree` (whose bound is set to 1) and `Int` (whose scope is set to 4). Setting the scope of `Int` to $k$ allows instances to have integer values from the set $\{0, 1, \ldots, k-1\}$[9].

The analyzer can enumerate all possible instances of an Alloy model. Since Alloy formulas are solved (via a translation) by off-the-shelf SAT solvers, the order of enumeration is essentially arbitrary and follows the particular algorithm implemented by the solver used. Figure 3-5 presents two of the instances the analyzer enumerates on simulating `Main`. Figure 3-6 (a) and (b) illustrate respectively `Instance 1` and `Instance 2`, which (as noted before) are isomorphic.

[9]Instances have non-negative integers.

```
Instance 1:
-----------
Int = { 0, 1, 2, 3 }

BinarySearchTree = { T0 }
Node = { N0, N1, N2 }

// fields of BinarySearchTree
root = { (T0, N0) }
size = { (T0, 3) }

// fields of Node
left = { (N0, N1) }
right = { (N0, N2) }
info = { (N0, 2), (N1, 1), (N2, 3) }


Instance 2:
-----------
Int = { 0, 1, 2, 3 }

BinarySearchTree = { T0 }
Node = { N0, N1, N2 }

// fields of BinarySearchTree
root = { (T0, N1) }
size = { (T0, 3) }

// fields of Node
left = { (N1, N0) }
right = { (N1, N2) }
info = { (N1, 2), (N0, 1), (N2, 3) }
```

Figure 3-5: Two Alloy instances that represent isomorphic trees.

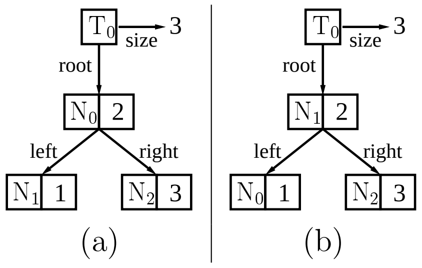

Figure 3-6: Isomorphic binary search trees.

# Chapter 4

# Specification Language

We define a veneer on the Alloy specification language [43] to write specifications of Java programs. The veneer introduces notation that translates into Alloy. This chapter describes our veneer, in particular:

- how we model mutation of state in Alloy (Section 4.1);
- how we model parameters and return value of a Java method in Alloy (Section 4.2); and
- our notation for writing specifications (Section 4.3).

Our model of mutable state lets us, through automatic translation, use the Alloy Analyzer for checking correctness of Java programs. It is worth pointing out that TestEra users need not be aware of the details of the underlying model and various other models (that have a relational basis) [45, 53, 66, 89] may be used instead. The users need to be familiar only with the notation, say for referring to the desired state. The translation automatically desugars TestEra specifications into the underlying model. Users, however, do need to be aware of the underlying semantics of the expressions in our notation. We point out the key aspects in which the semantics differ from Java semantics here.

At OOPSLA 2002 [53], we presented an annotation language similar in spirit to the veneer we present here.

## 4.1 Mutation

We model mutation of state simply by associating a distinct graph with each state. Mathematically, we treat non-array fields as ternary relations, each of which maps an object to an object in a given state.

We introduce `State` as a set of atoms:

```
sig State {}
```

Now, our model of a field `f` of type `T` declared in class `C` is the relation

```
f: C -> T -> State
```

where a tuple $(c, t, s)$ in `f` says that object $c$ has value $t$ in state $s$. This relation is modeled by the following signature declaration:

```
sig C {
  f: T -> State
}
```

Now, to access the value of field `f` of an object `o` of class `C` in state `s`, we write the expression `o.f.s`. It is worth noting that in this expression a relation on the right-hand-side of the dot operator has arity one.

Likewise, we update our model of arrays to include state. An array-type `T[]` is now modeled as:

```
sig T'' {
  length: Int !-> State,
  contents: Int -> T -> State
}
{
  all t: State | all i: Int {
    sole (i.contents).t // originally encoded using a cardinality marking
    i >= length.t => no (i.contents).t
  }
}
```

For the binary search tree example, we model the Java class and field declarations as the following Alloy signatures and fields:

```
sig BinarySearchTree {
  root: Node ?-> State,
  size: Int !-> State
}

sig Node {
  left: Node ?-> State,
  right: Node ?-> State,
  info: Int !-> State
}
```

The field `root` introduces a relation of type `BinarySearchTree -> Node -> State` (Figure 4-1 illustrates a valuation of this relation). A tuple $(b, n, s)$ denotes that the value of field `root` of tree $b$ in state $s$ is $n$. Recall that the cardinality marking '?' indicates that for each `BinarySearchTree` atom $b$ and each `State` atom $s$, there is *at most one* `Node` atom $n$ such that the tuple $(b, n, s)$ belongs to the relation `root`; similarly, the marking '!' indicates *exactly one*.

The model of state we have described is general in terms of the number of states. A method specification relates two states: the pre-state and the post-state. We model these states as follows. `State` consists of two distinct atoms `Pre` and `Post`, each modeled as a singleton subset of `State`:

```
static sig Pre extends State {}
static sig Post extends State {}

fact { Pre != Post }        // pre-state and post-state are distinct
fact { Pre + Post = State } // there are only two states
```

An artifact of our implementation is that we require the atoms `Pre` and `Post` to be distinct. This allows us to differentiate between values of an object's field in the pre-state and the post-state.

### 4.1.1 Other Models of Mutation

There are various ways of modeling mutation in a relational logic [45, 53, 66, 89]. Recall that we have introduced `State` as the last column in a relation. Instead, we could have introduced it as some other column, e.g., the first one [53]. That would require introducing, in the `State` signature, a field corresponding to each of the fields declared in the relevant classes, which would result in one signature that has all the fields. Even though this representation is not visible to the TestEra users, this approach for modeling mutation in general is not modular.

When a bound on the number of states is known a priori, an alternative model of mutation is to duplicate fields [66, 89]. As a particular illustration consider the case with two states. We can model a field `f` of type `T` declared in class `C` using the two relations `f1, f2:  C -> T`, where `f1` models field values in the pre-state and `f2` models the field values in the post-state. Notice that this approach avoids introducing ternary relations for (all) reference fields and therefore in cases where not all fields are updated in each state, it can provide more efficient checking by the Alloy Analyzer by generating smaller formulas with fewer boolean variables.

Our model of mutation that introduces a new state column in the relations is more elegant (since for example, it allows building compact specifications and treats state as a first-class entity) and presents a modular way to model mutation in general: each of the signature that models a class declares fields that model the corresponding fields of the class, and signature declarations from existing modules can simply be included in new modules. As stated before, from the perspective of the TestEra user, the details of the actual model of mutation used are irrelevant.

Fields do not have to be modeled as relations. Instead, we can model the heap using explicit references [45]. This approach also allows capturing object interactions, sharing and aliasing.

Even though these approaches offer similar expressivity, for testing or even static analysis of code using the Alloy Analyzer, it is worth investigating how the approaches compare with each other in performance.

## 4.2 Method Parameters and Return Value

Recall that the signature `State` consists of two distinct atoms `Pre` and `Post`, which are modeled as singleton sets. We model method parameters[1] and return values by introducing appropriate relations in `Pre` and `Post` as follows[2]. Consider a method `m` (declared in class `C`) with parameters `p1, ..., pn` of types `T1, ..., Tn` respectively. Let the type of the return value of `m` be `R`. We model each parameter `pi` with relation `pi: Pre -> Ti` and (for instance methods) the implicit parameter `this` with relation `This: Pre -> C`. We model the return value with relation `Result: Post -> R`.

As an illustration, consider the `remove` method for `BinarySearchTree`. We declare the `Pre` and `Post` states with fields that correspond to method parameters and return value:

```
static sig Pre extends State {
  This: BinarySearchTree,   // implicit parameter "this"
  i: Int                    // parameter "i"
}

static sig Post extends State {
  Result: Boolean           // return value
}
```

It is worth noting that the declaration of fields `This` and `i` in `Pre` is faithful to Java's call-by-value semantics. Since the references (or primitive values) that method arguments hold in the pre-state remain the same in the post-state, we do not need the model to allow fields that model the parameters to have different values in different states—the pre-state values are used throughout.

To illustrate an instance of an Alloy specification that models the pre-state and post-state of a method invocation, consider the valuation of the signatures `State`, `Pre`, `Post`, `BinarySearchTree` and `Node` and their fields (as declared above) given in Figure 4-1. Figure 4-2 illustrates this instance graphically, where we project relevant portions of the instance on (a) pre-state and (b) on post-state.

The user need not be familiar with how the parameters and return value of a method are modeled. We have packaged parameters together in the signature `Pre` as this allows simpler definition of isomorphisms among method inputs (Chapter 6). Next, we present notation that is automatically translated to the underlying model.

## 4.3 Notation for Writing Specifications

To allow users to talk about the pre-state in postconditions, we introduce the following notation: a field name followed immediately by a back-tick character '`' denotes

[1]Note capitalized `This` as a field of `Pre`.

[2]Following our treatment of single-dimensional arrays, if a method parameter is of an array-type `T[]`, we declare its type in `Pre` as `T''`. For a method return value that has an array-type, the treatment in `Post` is identical.

```
State = { S0, S1 }
Pre = { S0 }
Post = { S1 }

Boolean = { true, false }
Int = { 0, 1, 2, 3 }

BinarySearchTree = { T0 }
Node = { N0, N1, N2 }

// fields of Pre
This = { (S0, T0) }
i = { (S0, 2) }

// fields of Post
Result = { (S1, true) }

// fields of BinarySearchTree
root = { (T0, N1, S0),
         (T0, N0, S1) }
size = { (T0, 3, S0),
         (T0, 2, S1) }

// fields of Node
left = { (N1, N0, S0) }
right = { (N1, N2, S0),
          (N0, N2, S1) }
info = { (N1, 2, S0), (N0, 1, S0), (N2, 3, S0),
         (N0, 1, S1), (N2, 3, S1) }
```

Figure 4-1: An Alloy instance that represents the pre-state and post-state of an invocation of method `remove`.

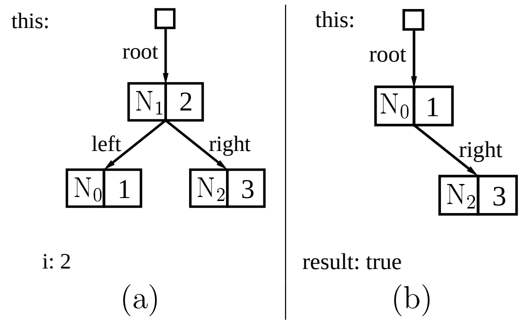

Figure 4-2: An example invocation of method `remove`. (a) Input tree and integer in pre-state of invocation. (b) The tree in post-state and the method return value.

traversal in the pre-state; all other traversals are in the default state, which is pre-state for preconditions and post-state for postconditions. We require all field names in an expression to follow the composition operator '`.`'. In particular, we require the explicit use of `this` in an expression.

Recall that we use `@result` to refer to the return value of a method. To allow easier parsing of expressions in our implementation, we refer to method parameters by prepending the formal name by '`@`'.

To allow users to conveniently access array elements[3], we write `o.a[i]` for the `i`th element of array `o.a`. To access the element in the pre-state, we write ``o.a`[`i]``.

As an illustration, the postcondition of `remove`:

```
// element removed correctly
this.root.*(left + right).info = this.root`.*(left` + right`).info` - @i

// result correct
@result = true <=> i in this.root`.*(left` + right`).info`
```

desugars to:

```
// element removed correctly
(Pre.This).(root.Post).*((left.Post) + (right.Post)).(info.Post) =
    (Pre.This).(root.Pre).*((left.Pre) + (right.Pre)).(info.Pre) - (Pre.i)

// result correct
(Post.Result) = true <=>
    (Pre.i) in (Pre.This).(root.Pre).*((left.Pre) + (right.Pre)).(info.Pre)
```

The class invariant desugars similarly: when the class invariant is used for test generation, the desugaring is with respect to the pre-state; when the class invariant is used to check structural invariants once the method returns, the desugaring is with respect to the post-state.

## 4.4 Comparison with JML

The Java Modeling Language [59] (JML) is a rich behavioral interface specification language designed for Java. Expressions in JML specifications are based on Java expressions. In addition, JML specifications may relate pre- and post-states or specify constraints on method return values, and support various design-by-contract constructs, including inheritance of contracts, and contracts that define exceptional behavior. JML also supports a range of quantifiers, including universal and existential quantifiers. The JML tool-set supports mainly runtime-assertion generation from JML formulas. JML has also been combined with JUnit—a popular test execution

[3] In Alloy, square-brackets are used for parameterized signatures and as an alternative way to write the join operator. We allow square brackets to be used for arrays only; the parameterized signature expression `S[T]` can be written as `S\T\`, and join is expressed using '`.`'.

and error reporting framework—to allow testing of programs on user-provided inputs [16].

Here, we present how some aspects of expressions in our veneer compare with those of JML expressions.

### 4.4.1 Evaluation in Pre-state

JML provides the construct `\old` to refer to the pre-state value of an expression. In particular, for an expression `e`, `\old(e)` evaluates to the value that `e` would evaluate to in the pre-state. The expression

```
this.root`.left`
```

that refers to the pre-state left child of the pre-state root node of `this` in our veneer could be expressed in JML as

```
\old(this.root.left)
```

However, the expression

```
this.root.left`
```

that refers to the pre-state left child of the *post*-state root node of `this` in our veneer cannot be directly expressed in JML. The expression `this.root.\old(left)` is ill-formed since the expression `left` cannot be evaluated on its own in the pre-state. Even though the JML language allows the use of temporary variables to state the desired expression, the tool-set does not support the evaluation of such expressions.

### 4.4.2 Relations and Navigation

Recall that an Alloy expression always denotes a relation. This is a key difference between Alloy and JML expressions. The Alloy expression `o.f` for object `o` and field `f` evaluates to a (possibly empty) set of objects. Following Java semantics, the same expression in JML evaluates to either the value `null` or an object reference.

JML provides the class `JMLObjectSet`, which can be used to build sets of objects. JML library defines this class using axioms in the spirit of algebraic specifications [38]. JML implements this class in Java; this implementation can be used for runtime checking.

### 4.4.3 Reachability

JML provides the construct `\reach`, akin to the transitive closure operator of Alloy, to express reachability. In particular, the expression `\reach(o)` for object `o` denotes the smallest `JMLObjectSet` that contains `o` and all objects that are accessible through any field of `o` and all objects that are accessible through any field of those objects and so on, recursively. However, the path expression

```
this.root.*(left + right)
```

is not expressible directly using the `\reach` construct since nodes also have the field `info` in addition to `left` and `right`. Such a set can, nonetheless, be constructed in JML by implementing a traversal algorithm that visits all nodes reachable via `left` or `right` fields only and returns a set of those nodes. But the construction would be more verbose. A veneer on JML could allow easier representation of such path expressions.

### 4.4.4 Comparison Summary

JML is conceptually founded on algebraic specifications. Our veneer is based on a simple first-order logic with relational operators, and is thus more in the tradition of semantic data modeling (now called "object modeling").

Our veneer and JML also differ in the evaluation of expressions that relate pre and post states. In our veneer, expressions are evaluated after both the pre and post states are available (in an abstract form). In JML, expressions that refer to pre-state are actually evaluated in the pre-state and their values are stored to be accessed later in the post-state, in which the expressions referring to post-state are evaluated. Thus in our veneer, we can state expressions that freely interleave pre and post states, whereas the JML tool-set requires that evaluation of any expression enclosed in `\old` must be possible in the pre-state itself.

Another difference is that JML allows Java expressions and evaluates them according to Java semantics. Even though JML does not support general path expressions, a veneer can be defined to provide the support.

JML is a full-fledged specification language and provides a wide range constructs, such as ghost and model fields and inheritance of specifications, that our veneer does not support. The JML tool-set, however, mainly targets runtime assertion generation and does not provide some of the automatic analysis capabilities, including checking of specifications themselves and test generation from input constraints, that our veneer allows. We discuss in Chapter 9, a more recent approach [13] that enables automatic test generation from JML preconditions.

# Chapter 5

# Translations

In TestEra, structures are described in an abstract logic. These descriptions are used to generate concrete tests as Java objects. To bridge the gap between the abstract domain and the concrete domain, TestEra provides algorithms that translate between the domains. These translations provide a way to bring fully automatic analysis to Java programs that manipulate structures. It is worth noting that the these translations need not be used only for testing programs. The same algorithms can be used for implementing various other techniques that ensure software reliability, say by monitoring conformance or by maintaining crucial invariants at runtime.

In the context of testing, TestEra implements various algorithms in addition to concretization and abstraction translations of data. To enable using the Alloy Analyzer [46] for test generation and correctness checking, TestEra creates Alloy [43] specifications that consist partly of the user-provided constraints and partly of automatically generated declarations that TestEra builds from given Java classfiles. To enable test execution, TestEra creates a Java driver that appropriately invokes the method to test.

The chapter describes the basic architecture of TestEra and its key algorithms. To test a method, TestEra performs the following three steps:

1. Create Alloy specifications for test generation and correctness checking;

2. Create Java driver for translating data between Alloy instances and Java objects; and

3. Generate tests, execute method and check correctness.

We also describe our prototype implementation in this chapter.

TestEra was first presented at ASE 2001 [66]. A more detailed account is to appear at the Journal of Automated Software Engineering [52].

## 5.1 Architecture

Figure 5-1 shows the main components of TestEra. Boxes represent processes that generate inputs and check correctness; ellipses represent programs and specifications

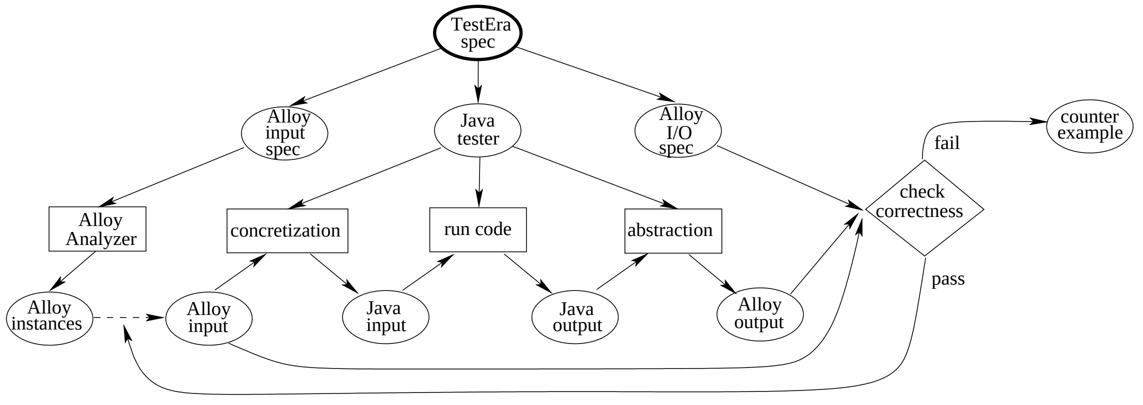

Figure 5-1: Basic TestEra framework. The user provides "TestEra spec", which consists of the input constraint, the correctness criteria, the name of the method to test and its implementation, and the scope. All subsequent steps are automatic.

that are given by the user, and inputs and outputs that are generated and used to check correctness. A TestEra specification for the method to test provides the method declaration (i.e., the method return type, name, and parameter types), the name of the Java classfile (or sourcefile) that contains the method body, the class invariant, the method precondition, the method postcondition, and a bound on the input size. In principle, the method postcondition is not required. If the post-condition is not given, TestEra reports inputs that raise exceptions; these inputs are also reported when actual post-conditions are given. Java assertions that may be present in the code are also checked automatically.

Given a TestEra specification, TestEra creates three files. Two of these files are Alloy specifications: one specification is for generating inputs and the other specification is for checking correctness. The third file consists of a Java test driver, i.e., code that translates Alloy instances to Java input objects, executes the Java method to test, and translates Java output objects back to Alloy instances.

TestEra's automatic analysis proceeds in two phases:

- In the first phase, TestEra uses the Alloy Analyzer to generate all nonisomorphic instances of the Alloy input specification.

- In the second phase, each of the instances is taken in turn. It is first *concretized* into a Java test input for the method. Next, the method is executed on this input. Finally, the method's output is *abstracted* back to an Alloy instance. TestEra uses the Alloy Analyzer to check if the output Alloy instance and the original input Alloy instance satisfy the constraints in the input/output (or correctness) specification. If the check fails, TestEra reports a counterexample. If the check succeeds, TestEra uses the next Alloy input instance for further testing.

These phases can be interleaved: for each input that TestEra generates, it can execute the method to test and check the resulting input/output pair for correctness.

## 5.2 Building Alloy Specifications

TestEra parses Java classfiles and builds Alloy specifications by combining signature declarations with Alloy functions that express method pre and postconditions that are given by the user. These specifications explicitly introduce state following the approach we described in Section 4.1.

### 5.2.1 Input Generation Specification

The algorithm for creating the Alloy specification that represents the method's precondition and is used for test generation proceeds in two steps. Recall that the precondition expresses a constraint on the pre-state. Therefore, in the first step the algorithm introduces the signature declaration for the pre-state. In the second step the algorithm creates an Alloy function that represents the method precondition. The precondition that is given in the veneer is appropriately translated to an Alloy formula that represents the input constraint.

To test a method $m$ in a jar file $jar$, where $m$'s input constraint is $pre_m$, the first step proceeds as follows. Signature (and field) declarations are created for each of

- The special `State` signature (see Section 4.1);
- Pre-state, where there is a field corresponding to each parameter of $m$;
- Every single-dimensional array type that is a field type or a parameter type or the result type of $m$;
- Every class $C$ in $jar$ (which does not have a library spec). For each class the fields of the corresponding signature are declared as follows. For a field $f$ declared in $C$;
  - if $f$ is of a non-array reference type $T$, declare the field as `f: T ?-> State`;
  - if $f$ is of a single-dimensional array type $T[]$, declare the field as `f: T'' ?-> State`;
  - if $f$ is of primitive type $T$, declare the field as `f: T !-> State`.

In the second step the function declaration for `InputConstraint` is created such that:

- The parameter list of the function is empty;
- The body of the function is $pre_m$, where we replace every occurrence of
  - field `f` by `(f.Pre)`;
  - parameter `@p` of $m$ by `(Pre.p)`;
  - array access `o.f[i]` by `i.(o.(f.Pre).(contents.Pre))`.

```
module testera/mutation/State

sig State {}
```

Figure 5-2: Signature declaration for `State`.

Since the precondition expresses a constraint on the pre-state, the algorithm systematically introduces dereferencing in the pre-state to translate the veneer into Alloy.

Our TestEra implementation generates a separate Alloy module for each signature (that corresponds to a class) and uses Alloy's facility of re-using existing specifications to include modules in the current specification using the `open` command. `State` is a pre-defined signature in the module `testera/mutation/State` (Figure 5-2).

To illustrate generation of Alloy specifications, recall the binary search tree example of Section 1.2 and the method `remove`. The method had the following declaration:

```
  boolean remove(int i) { ... }
```

The class invariant of `BinarySearchTree` was given as:

```
 // binary tree
 all n: this.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
 }

 // elements in correct order
 all n: this.root.*(left+right) {
   all nl: n.left.*(left + right) | nl.info < n.info
   all nr: n.right.*(left + right) | nr.info > n.info
 }

 // size ok
 this.size = #root.*(left + right)
```

This de-sugars to:

```
 // binary tree
 all n: this.(root.Pre).*((left.Pre) + (right.Pre)) {
   n !in n.^((left.Pre) + (right.Pre)) // no directed cycle
   sole n.~((left.Pre) + (right.Pre))  // at most one parent
   no n.(left.Pre) & n.(right.Pre)     // distinct children
 }

 // elements in correct order
 all n: this.(root.Pre).*((left.Pre)+(right.Pre)) {
   all nl: n.(left.Pre).*((left.Pre) + (right.Pre)) | nl.info < n.info
   all nr: n.(right.Pre).*((left.Pre) + (right.Pre)) | nr.info > n.info
```

```
}

// size ok
this.(size.Pre) = #this.(root.Pre).*((left.Pre) + (right.Pre))
```

To generate inputs for the method `remove`, TestEra builds the specification that consist of the modules shown in Figures 5-3, 5-4, and 5-5.

### 5.2.2 Correctness Checking Specification

The algorithm for creating the Alloy specification that represents the method's postcondition and is used in checking the method's correctness proceeds in two steps. Recall that the postcondition relates the pre-state and the post-state. Therefore, in the first step the algorithm introduces the signature declaration for the post-state. In the second step the algorithm creates an Alloy function that represents the method postcondition. The postcondition that is given in the veneer is appropriately translated to an Alloy formula.

To test a method $m$ in a jar file $jar$, where $m$'s postcondition is $post_m$, the first step proceeds as follows. Signature (and field) declarations are included for

- Each signature that was declared in the input generation specification
- Post-state, where there's a field corresponding to the return type of $m$ if $m$ is not `void`

In the second step, the function declaration for `MethodOk` is created as follows.

- The function has no parameters
- $post_m$ is the body of the function, where we replace every occurrence of
  - field `` f` `` (i.e., field that has a backtick) by `(f.Pre)`
  - field `f` (i.e., field that does not have a backtick) by `(f.Post)`
  - array element `` o.f`[`i] `` by `i.(o.(f.Pre).(contents.Pre))`
  - array element `o.f[i]` by `i.(o.(f.Post).(contents.Post))`
- method parameter `@p` by `(Pre.p)`
- method return value `@result` by `(Post.result)`

Since a postcondition may express a constraint that relates pre-state and post-state, the algorithm introduces dereferencing in the appropriate state following the convention that backticks identify pre-state components.

For checking correctness of `remove`, TestEra builds the Alloy specification given in Figure 5-6.

To generate input instances TestEra simulates `InputConstraint` using the Alloy Analyzer for the input scope[1]. To check correctness, TestEra uses the analyzer

[1]The scope of `State` for input generation is fixed at 1.

```
module testera/examples/BinarySearchTree

open testera/mutation/State
open testera/examples/Node

sig BinarySearchTree {
  root: Node ?-> State,
  size: Int !-> State
}
```

Figure 5-3: Signature declaration for `BinarySearchTree`.

```
module testera/examples/Node

open testera/mutation/State

sig Node {
  left: Node ?-> State,
  right: Node ?-> State,
  info: Int !-> State
}
```

Figure 5-4: Signature declaration for `Node`.

```
module testera/examples/BinarySearchTree/remove_input

open testera/mutation/State
open testera/examples/BinarySearchTree
open testera/examples/Node

static sig Pre extends State {
  This: BinarySearchTree,
  i: Int
}

fun BinarySearchTree::ClassInvariant() { ... }

fun InputConstraint() { (Pre.This)..ClassInvariant() }
```

Figure 5-5: Alloy specification for generating inputs for `remove`.

```
module testera/examples/BinarySearchTree/remove_output

open testera/mutation/State
open testera/examples/BinarySearchTree
open testera/examples/Node

static sig Pre extends State {
  This: BinarySearchTree,
  i: Int
}
static sig Post extends State {
  Result: Boolean
}

fact { Pre != Post and Pre + Post = State }

fun BinarySearchTree::ClassInvariant() { ... }

fun MethodOk() {
  // class invariant preserved
  (Pre.This)..ClassInvariant()

  // element removed correctly
  (Pre.This).(root.Post).*((left.Post) + (right.Post)).(info.Post) =
    (Pre.This).(root.Pre).*((left.Pre) + (right.Pre)).(info.Pre) - (Pre.i)

  // result correct
  (Post.Result) = true <=>
    (Pre.i) in (root.Pre).*((left.Pre) + (right.Pre)).(info.Pre)
}
```

Figure 5-6: Alloy specification for checking correctness of `remove`.

to check whether the given input/output pair satisfies the constraints expressed in `methodOk` for the output scope[2].

## 5.3 Translations

We next discuss the test driver that TestEra generates to test the specified method. A test driver consists of Java code that performs abstraction and concretization translations and invokes the method to test. The translations are generated fully automatically if the method specification is given at the representation level of method inputs. If the specification introduces a level of data abstraction, the translations have to be

[2]The scope of `State` for correctness checking is fixed at 2.

manually provided.

Concretization, abbreviated `a2j`, translates Alloy instances to Java objects (or data structures). Abstraction, abbreviated `j2a`, translates Java data structures to Alloy instances.

### 5.3.1 Concretization

Concretization `a2j` operates in two stages. In the first stage, `a2j` creates, for each atom in the Alloy instance, a corresponding object of the Java classes, and stores this correspondence in a map. In the second stage, `a2j` establishes the relationships among the Java objects created in the first stage and builds the actual data structures.

Figure 5-7 describes a generic concretization algorithm `a2j`. The algorithm takes as input an Alloy instance and returns an array of Java objects that represent a test input and maps that are used in checking correctness. The algorithm maintains two maps to store correspondence between Alloy atoms and Java objects: `mapAJ` maps atoms to objects and `mapJA` maps objects to atoms. In the first step, `a2j` creates Java objects of appropriate classes for each atom in the instance[3]. In this step objects of single-dimensional array-types are also appropriately initialized by setting their lengths and contents. In the second step, `a2j` sets values of objects according to tuples in the input relations; notice that all tuples represent values in the pre-state. Finally, `a2j` builds an array of objects that represents a test input: for an instance method, `input[0]` represents the object on which to invoke the method, `input[1]` represents the first declared parameter, and so on.

### 5.3.2 Abstraction

Abstraction `j2a` operates by traversing, in the post-state, the structures that were input to the method or returned by it, and creating relations of the Alloy instance that represents the structures in the post-state.

Figure 5-8 describes a generic abstraction algorithm `j2a`. The algorithm takes as input the method return value (`result`), `imap` constructed during concretization and the instance that was concretized, and adds tuples to this instance to build an instance that represents the corresponding input/output pair. The algorithm adds the output component to `io` by traversing the structures referenced by `result` and inputs in `imap` (in the post-state). This traversal is a simple work-list algorithm that tracks the set of visited objects. For each object that is visited for the first time, `j2a` adds tuples to `io` according to field values of that object. If an object that did not exist in the pre-state is visited, a new atom is created and `imap` updated; this accounts for cases when a method allocates new objects. Single-dimensional array objects are handled appropriately by updating the appropriate `length` and `contents` relations.

[3]For atoms that represent primitive values, TestEra uses library classes testera.primitive.*; for other atoms TestEra invokes appropriate constructors.

```
InputsAndMaps a2j(Instance a) {
  Map mapAJ = new HashMap(), mapJA = new HashMap();

  // for each atom create a corresponding Java object

  // non-array types
  foreach (sig in a.sigs() such that sig represents a non-array type)
    foreach (atom in sig) {
      SigClass obj = new SigClass();
      mapAJ.put(atom, obj);
      mapJA.put(obj, atom);
    }

  // 1-dim array types
  foreach (sig in a.sigs() such that sig represents a 1-dim array type)
    foreach (atom in sig) {
      int length = a.getLength(atom, Pre);
      SigClass[] obj = new SigClass[length];
      for (int i = 0; i < length; i++)
        obj[i] = mapAJ.get(a.getContent(atom, i, Pre));
      mapAJ.put(atom, obj);
      mapJA.put(obj, atom);
    }

  // establish relationships between created Java objects
  foreach (rel in a.relations())
    foreach (<x,y> in rel::Pre)
      setField(mapAJ.get(x), rel, mapAJ.get(y));

  // set inputs
  Object[] inputs = new Object[a.numParams(repOk)];
  for (i = 0; i < a.numParams(repOk); i++)
    inputs[i] = mapAJ.get(a.getParam(repOk, i));

  result = (inputs, mapAJ, mapJA);
}
```

Figure 5-7: Concretization algorithm `a2j`.

This completes our description of the key algorithms TestEra provides to automatically generate Alloy specifications and perform abstraction and concretization translations.

```
j2a(Object result, InputsAndMaps imap, Instance io) {
  Set visited;
  List worklist = result + imap.inputs;

  while (!worklist.isEmpty()) {
    Object current = worklist.getFirst();
    if (!visited.add(current)) continue;

    if (current is not of an array type)
      foreach (field in getFields(current)) {
        Object to = getValue(field, current);
        if (to == null) continue;
        Atom toAtom = !mapJA.containsKey(to) ? getNewAtom(to, imap) :
                                               imap.mapJA.get(to);
        io.addTuple(field, imap.mapJA.get(current), toAtom, Post);
        if (!visited.contains(to)) worklist.add(to);
      }

    if (current is of a 1-dim array type) {
      int length = Array.getLength(current);
      Atom fromAtom = imap.mapJA.get(current);
      io.addTuple(length, fromAtom, length, Post);
      for (int i = 0; i < length; i++) {
        Object to = Array.get(current, i);
        Atom toAtom = !mapJA.containsKey(to) ? getNewAtom(to, imap) :
                                               imap.mapJA.get(to);
        io.addTuple(contents, fromAtom, i, toAtom, Post);
      }
    }
  }
}
```

Figure 5-8: Abstraction algorithm `j2a`.

### 5.3.3 Traversing the Heap

The abstraction algorithm traverses the entire part of the heap reachable from method inputs, following instance fields, and assigns values to relations accordingly. This traversal is oblivious of the postcondition and therefore could result in needlessly abstracting parts of heap that do not influence the evaluation of the postcondition. A more efficient translation can (statically) analyze the postcondition to identify relevant parts of the heap and focus the abstraction on those parts.

If some fields are irrelevant for a particular testing purpose (say because they are not used in the code being tested), users can instruct TestEra to ignore those fields: (1) the underlying Alloy specifications then do not represent those fields, which reduces the size of boolean formulas and allows faster test generation and correctness checking; (2) the translation algorithms ignore the values of those fields (i.e., they are set to `null` or appropriate default primitive values).

Recall that Alloy formulas may include the transpose operator. Notice that the abstraction algorithm builds relations using a *forward* traversal of the heap and thus assumes that the components of the heap relevant to evaluating given formulas are reachable from the given root-set. This is particularly worth noting when using the transpose operator in the case when TestEra is explicitly directed to ignore certain fields. The default is to model all fields.

## 5.4 Implementation

Our prototype implements these algorithms in Java. For bytecode parsing we use the ByteCode Engineering Library [26]. For solving SAT formulas we use the mChaff [71] solver. We have also explored the relsat [7] solver but experiments on enumerating a variety of benchmark data structures show that mChaff outperforms relsat (Section 7.1).

TestEra can currently be used only by providing command line arguments: to test a method the user provides the method signature, the name of the class that declares it, (filenames for text files containing) the class invariant, method precondition and method postcondition, and the scope. To enable testing programs that use interfaces, TestEra allows users to provide *actual* (concrete) types for fields and method parameters and return value; these types override the declared types and are used in test generation and correctness checking; the default is to use the declared types. For example, to test `java.util.TreeMap` where keys in nodes need to be `Comparable`, we direct TestEra to use `Integer` as the actual type.

TestEra provides a variety of libraries that can be reused when generating specifications. These libraries include models of several data structures (some of which are discussed in Chapter 7) and models of non-negative integers and booleans, which were not supported by default in the version of Alloy Analyzer that was available at the time of our experiments.

### 5.4.1 Translation Optimizations

In our prototype, boolean solutions generated by the SAT solver are translated to Alloy instances, which are then concretized into Java tests using the Alloy API. It really is the boolean solutions that determine what Java tests are generated and these boolean solutions can be directly concretized into tests. Therefore, a more efficient way to perform concretizations is to generate code that performs the translation for a given benchmark and a given scope. Similarly, to check correctness, we could directly translate Java outputs into boolean assignments and use SAT to check satisfiability.

Therefore, a more efficient approach than the one we have implemented is to use the Alloy Analyzer for compiling first-order logic specifications into propositional formulas and then directly using SAT together with (automatically generated) specialized translations for test generation and correctness checking. We can optimize this further by using algorithms that generate translations that do not use Java's reflection.

### 5.4.2 Beyond Java Programs

Even though TestEra is implemented in Java, it can be used to test programs that are implemented in other languages, such as C++. For example, a program that takes as input a text file can be tested by generating files in appropriate format. The Galileo case-study (Section 7.2.2) illustrates such a scenario. When testing such programs with TestEra, the abstraction and concretization translations need to be written manually because the automatic translations that TestEra currently implements can parse only Java classfiles.

# Chapter 6

# Nonisomorphic Generation

This chapter explains our technique for nonisomorphic generation. We first give a definition [13] of nonisomorphism among structures in a Java program. Next, we outline the default support [79] that the Alloy Analyzer provides for symmetry breaking, which can be used to enumerate nonisomorphic structures but is inefficient. Finally, we describe our technique that efficiently generates exactly nonisomorphic inputs.

Our technique for generating nonisomorphic inputs appears in SAT 2003 [54].

## 6.1 Object Identities

In object-oriented languages, each object is assigned an identity that can be used for comparison with other object identities (and the constant `null`). This comparison is typically implemented by comparing memory addresses. In safe languages, such as Java, an actual address, however, cannot be directly used in computation. This usage contrasts with that of the memory addresses in C/C++, where addresses may freely be used in computation. Even though a Java program does not have direct access to the address of an object, the program behavior may still depend on the actual identity. This is possible through the use of `System.identityHashCode(...)`. This method is rarely used explicitly by programmers, but hashtables often do computations based on it. Since the hashcode of an object can take any integer value, it is simply not feasible to systematically deal with the source of nondeterminism that hashcodes represent.

For testing Java programs, we assume that object identities are not used in any operation other than comparisons among identities. Notice that test generation (at the representation level or at the abstract level) cannot generally dictate exactly what identities objects have. For example, executing the construction sequence

```
BinarySearchTree bst = new BinarySearchTree();
bst.add(1);
bst.add(2);
bst.add(3);
```

twice will generate two structures that are isomorphic but not necessarily identical. Similarly, writing a structure to disk using Java's object serialization and reading

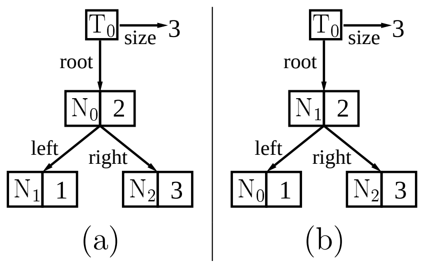

Figure 6-1: Isomorphic trees.

it back twice generates isomorphic but not necessarily identical structures. The assumption guarantees that the same test produces the same result on any number of runs of a program.

## 6.2 Definition

Before we define the notion of structural isomorphism, let us illustrate some of the concepts we use in the definition. Figure 6-1 shows two binary search tree structures. Both structures consist of the set $\{T_0\}$ of tree objects, the set $\{N_0, N_1, N_2\}$ of node objects and the set $\{1, 2, 3\}$ of primitive values.

We say a structure is *rooted* if it has a unique object $o$ such that all objects (and primitive values) in the structure are reachable via 0 or more field traversals from $o$; the object $o$ is called the *root*. For example, the tree in Figure 6-1(a) is rooted at $T_0$.

Let $S$ be a structure rooted at $s$. We write $R(S, s)$ for the set of all objects reachable from $s$ within $S$. If the field $f$ of object $o \in R(S, s)$ has the value $v$, we write $o.f \;\; =_S \;\; v$. We write *fields*$(o)$ to denote the set of all fields that are declared in the class that $o$ belongs to. For example, if $S$ is the structure in Figure 6-1(a), $R(S, T_0) = \{T_0, N_0, N_1, N_2\}$ and *fields*$(T_0) = \{$`root`, `size`$\}$.

Let $O_1, \ldots, O_n$ be sets of objects from $n$ classes. Let $O = O_1 \cup \ldots \cup O_n$ and let $P$ be the set containing all values of primitive types. Let $S$ and $T$ be two structures rooted at $s$ and $t$ respectively and consisting only of objects from $O$ and primitive values. $S$ and $T$ are *isomorphic* if and only if there exists a permutation $\pi$ 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 $\pi(s) = t$ and

$$\forall o \in R(S, s) \;\cdot\; \forall f \in \mathit{fields}(o) \;\cdot\; \forall v \in O \cup P \;\cdot\;\; o.f =_S v \;\Leftrightarrow\; \pi(o).f =_T \pi(v).$$

The structures in Figure 6-1 are isomorphic: witness the permutation $(T_0T_0)(N_0N_1)(N_1N_0)(N_2N_2)(11)(22)(33)$.

Figure 6-2 shows 15 structures that represent all nonisomorphic binary search trees with up to 3 nodes (where `info` field takes a value between 1 and 3).

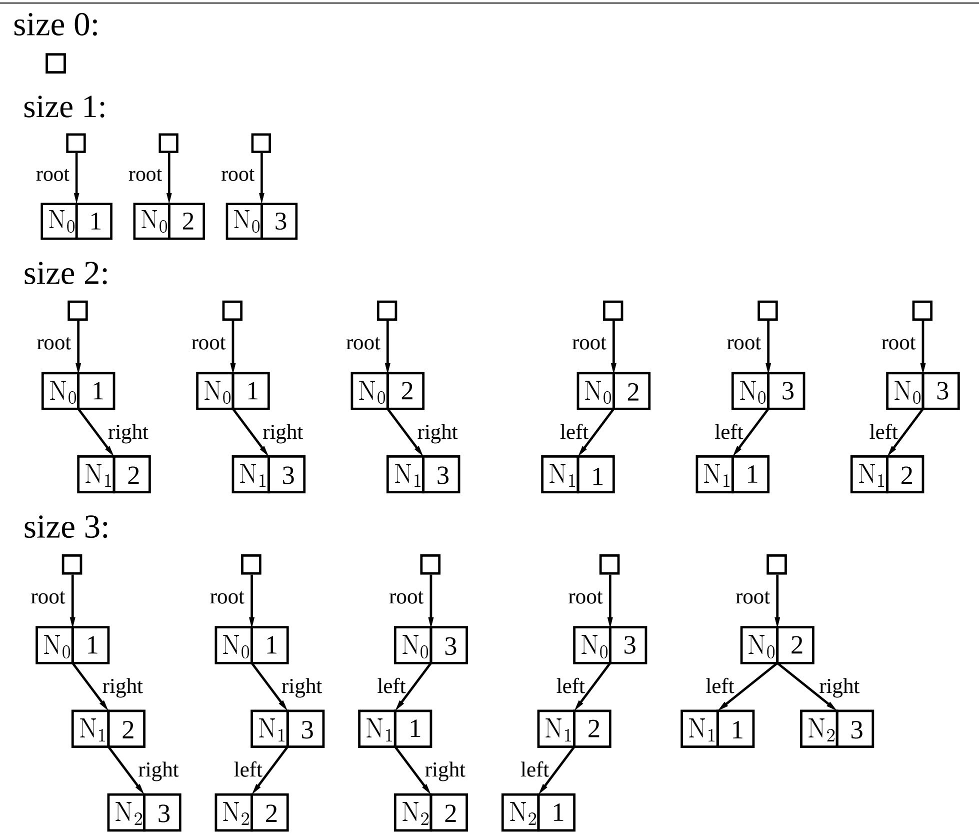

Figure 6-2: Trees generated for scope three. The identity of the tree object is omitted for clarity.

### 6.2.1 Method Inputs

We next describe how the definition of isomorphism among structures extends to the definition of isomorphism among method inputs, where each input can be a tuple consisting of several structures. Consider, for example, generating nonisomorphic inputs for the `remove` method in `BinarySearchTree` (Section 1.2). Using a scope of 3 (i.e., 3 nodes and field `info` and parameter `i` taking a value between 1 and 3), there are 45 nonisomorphic inputs: each input is a pair $(t, i)$ where $t$ is one of the 15 trees shown in Figure 6-2 and $i \in \{1, 2, 3\}$.

For a method `m` with $n$ formal parameters `f1, ..., fn` of types `T1, ..., Tn` respectively, isomorphism among method inputs is defined as isomorphism among the structures that are represented by the class:

```
class Input_m {
  T1 f1;
  ...
  Tn fn;
}
```

where the precondition of method `m` defines the structural constraints for instances of class `Input_m`. For any structure that represents an instance of `Input_m` we can generate an input for `m`. Nonisomorphic instances of `Input_m` define nonisomorphic inputs for method `m`.

Recall our model of method parameters (Section 4.2). The construction of signature `Pre` models precisely `Input_m`. It is worth pointing out that this approach handles aliasing among method parameters. For example, if a class method merges two input lists, the model allows both the inputs to reference the same list. Of course, if desired, aliasing can be ruled out by explicitly ruling it out in the method precondition.

## 6.3 Default Support in Alloy Analyzer

The Alloy Analyzer adapts the symmetry-breaking predicates of Crawford et al. [25] for faster solving. The original boolean formula is conjoined with additional clauses in order to produce only a few instances from each isomorphism class [79]. The basic idea in this technique is to take a structure's binary encoding and fix an ordering of the bits in the encoding. This induces a strict lexicographical ordering on the set of all structures. Then, by constructing a symmetry-breaking predicate that is true only on the lexicographically smallest structure in each isomorphism class and conjoining the original formula with this predicate, generation is restricted to nonisomorphic structures. Even though the analyzer allows generating such a *complete* symmetry-breaking predicate, doing so in general is NP-complete [25] and infeasible in practice because the resulting formulas become very large even for small scopes and choke the solvers.

As an example of how the analyzer constructs its symmetry-breaking predicates, consider generating all non-isomorphic sets. The following specification defines a set:

```
module set_def

sig S {}
```

Let us fix the scope to 2, i.e., in any instance of `S`, atoms are selected from say $\{s_0, s_1\}$. A simple boolean encoding would represent an instance as two bits $b_0b_1$, where $b_i = 1 \;\Leftrightarrow\; s_i \in S$ in the corresponding instance. The propositional formula (without symmetry-breaking) that corresponds to the specification `set_def` is just `true`. Let's fix the ordering $0 < 1$ on bits and induce the lexicographic ordering $00 < 01 < 10 < 11$ on all solutions. The only (non-identity) permutation, which permutes $b_0$ and $b_1$, defines the symmetry-breaking constraint, which encodes

$$b_0b_1 \leq b_1b_0$$

Thus, the propositional formula that represents symmetry-breaking boils down to

$$b_0 \Rightarrow b_1$$

Conjoining this formula with the original formula (i.e., `true`) and enumerating solutions of the resulting formula generates three nonisomorphic solutions: $\{\}$, $\{s_1\}$, $\{s_0, s_1\}$, which represent the only three nonisomorphic sets that contain at most two elements.

To enable practical enumeration, the Alloy Analyzer allows users to control the extent to which symmetries are broken. Although the analyzer can be set to enumerate exactly nonisomorphic instances, the resulting formulas tend to grow large, which slows down enumeration.

Moreover, the built-in symmetry breaking of the analyzer (without explicit addition of further constraints) breaks isomorphisms with respect to the *whole* instance. Indeed, isomorphism, in general and in Alloy, is defined with respect to the whole instance and not with respect to just the component reachable from a root. For test generation, however, we are only interested in the component that is reachable from a given root and thus require symmetry-breaking with respect to this particular component.

We next describe how to manually construct Alloy formulas that completely eliminate isomorphs (with respect to desired components) and also provide efficient enumeration for a variety of benchmark data structures [54].

## 6.4 Efficient Symmetry Breaking

The basic idea behind our technique is to use total orders and define a *canonicalization* of the underlying data structure. We conjoin the original Alloy formula that defines structural constraints with a symmetry-breaking formula that is also written in Alloy (and not at the level of propositional logic) and requires the generated structures to be in the canonical form for their respective isomorphism classes. Unlike the built-in symmetry breaking of the analyzer, this technique provides domain specific symmetry-breaking, which generates fewer clauses to break symmetries, but at the cost of requiring the user to add symmetry-breaking predicates manually.

It is worth pointing out that our aforementioned definition of isomorphism is for Java structures. The symmetry-breaking formulas, however, are written in Alloy. Our model of state (that views objects as atoms and fields of objects as relations) allows the definition of isomorphism between rooted structures in the Java domain to trivially carry over to the Alloy domain.

### 6.4.1 Total Orders in Alloy

The analyzer's standard library of models provides a polymorphic signature `Ord[t]`, which we use in writing our symmetry-breaking formulas. Each instantiation of `Ord` with some Alloy signature `t` imposes a total order on the elements in `t`. These elements no longer are indistinguishable, and the analyzer does not break symmetries further on that set. And the analyzer considers only one total order, instead of $(\#t)!$ possible total orders. In addition to the definition of total order, the analyzer's standard library also provides several Alloy functions for totally-ordered sets. To write constraints for nonisomorphic generation, we use the functions `OrdPrevs` and `OrdNexts`, which take an element as input and return a set of elements: `OrdPrevs` (respectively `OrdNexts`) returns the sets of all elements that are smaller (respectively larger) than the given element. The function `OrdFirst` returns the first element in the given signature.

### 6.4.2 Adding Symmetry-breaking Formula

Our symmetry-breaking Alloy formula defines a *traversal* that deterministically visits all reachable nodes and requires that the nodes are visited in an a priori fixed order. Such a traversal has the effect of implicitly defining a mapping from a structure to its canonical representation and restricting generation to only those structures that map to themselves.

For structures that can be represented using edge-labeled directed rooted graphs where each edge connects a node to exactly one node[1], we build the symmetry-breaking Alloy formula by:

- Defining a linear order on atoms;
- Defining a traversal (from the root) that *linearizes* the whole instance, i.e., deterministically visits all the reachable nodes; since the traversal is deterministic, it typically defines an order in which to traverse the fields;
- Requiring that all these nodes (expect for those that represent primitive values) are visited (for the first time) in the pre-defined linear order.

Then, any structure for which the formula does not hold will not be generated.

#### Example

As a simple illustration, consider enumerating nonisomorphic singly-linked lists of nodes (that contain no data) as defined by:

```
class List {
    Entry header;

    static class Entry {
        Entry next;
    }
}
```

Consider generating lists using the nodes $\{E_0, E_1, E_2\}$. To build the symmetry-breaking formula, we first order the nodes say $[E_0, E_1, E_2]$. Next, we define the traversal to start at the root node and follow the `next` field. We capture the essence of the constraint that the traversals visits the nodes in order as:

```
// uses library function to instantiate Ord[Node]
// header is the first node (or null)
header in OrdFirst(Entry)
```

[1]These graphs can represent at the concrete level any structure that can be implemented in Java.

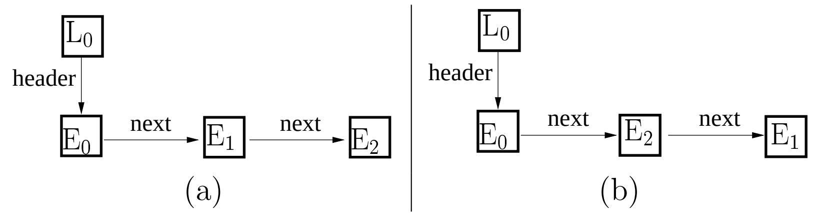

Figure 6-3: Isomorphic lists.

```
// traversal visits all nodes in order
all n: root.*next | some n.next => n.next = OrdNext(n)
```

Expressing these constraints allows generation of the list illustrated in Figure 6-3(a) but rules out generation or the list illustrated in Figure 6-3(b) since the `next` node of $E_0$ can only be $E_1$.

## Argument for Correctness

We next argue informally why our technique is correct. The argument has three parts:

- Soundness, i.e., no invalid structures are generated. Symmetry-breaking formula is conjoined with the original structural constraints therefore any structure that is generated satisfies the original structural constraints.

- Completeness, i.e., at least one valid structure from each isomorphism class is generated. The proof of completeness proceeds by contradiction. Assume that the algorithm does not generate any structure from an isomorphism class $C$ of valid structures. Let structure $S \in C$ have $n$ nodes $\{N_1, \ldots, N_n\}$ and the node ordering be $N_1 < \ldots < N_n$. Let the traversal of $S$ visit the nodes in the order $[N_{i_1}, \ldots, N_{i_n}]$ (where $1 \leq i_1, \ldots, i_n \leq n$). Consider now the structure $S'$ that is generated by applying the permutation $(N_{i_1}N_1)(N_{i_2}N_2)\ldots(N_{i_n}N_n)$ to $S$. Since the traversal algorithm is deterministic, nodes in $S'$ are visited in the order $[N_1, \ldots, N_n]$, which is the allowed order. Since $S'$ is isomorphic to $S$ and $S$ is valid, $S'$ is also valid. (This is because the structural constraints treat node identities as atoms.) Thus $S'$ is generated. Contradiction.

- Optimality, i.e., exactly one structure from each isomorphism class is generated. For any two distinct structures that are isomorphic, the traversal algorithm cannot visit all reachable nodes in the same order in both the structures because the traversal algorithm is deterministic and an edge connects a node to exactly one node. Since nodes are totally ordered, the traversal order of nodes in one of the structures must be lexicographically smaller than that for the other structure that will, therefore, not be generated.

Our traversal technique is valid only for rooted directed graphs where an edge connects a node to exactly one node (as is the case with our model of the heap of a Java program). For arbitrary graphs where an edge may connect a node to a *set* of nodes, it is possible to define a deterministic traversal, for example traverse in a depth first fashion, breaking choices among nodes by picking the smallest one (according to node ordering) that is not visited so far. However, such a deterministic traversal on two isomorphic (but not identical) graphs may visit the nodes in the same order and may therefore allow generating both. This violates optimality. The arguments for soundness and completeness still remain valid.

In the context of test generation, notice that an instance generated by the Alloy Analyzer may have components that are not reachable from the root(s). For example, an instance that represents a binary search tree may have edges between nodes that are not connected to the root and are therefore irrelevant to the concretization of the actual test. To restrict generation to instances that are nonisomorphic on the component reachable from the root we add to, symmetry breaking constraints, constraints that force fields of disconnected objects to take pre-defined fixed values. Hence any two instances that have isomorphic components reachable from the root, have identical components that are not reachable from the root.

### 6.4.3 Example

Let's consider defining a canonicalization for our binary search tree specification (which is reproduced in part in Figure 6-4). This specification represents one binary search tree—the scope of `BinarySearchTree` is set to 1. Since for any instance we are interested only in the component reachable from root we define symmetry breaking for that component in the following steps:

- Declare `Node` to be totally ordered;
- Order the fields [`root`, `left`, `right`];
- Define a traversal according to the ordering; in particular, when there is an option of which outgoing to edge follow, traverse with respect to the field ordering; and
- Require that the traversal visits (new) nodes in the expected order.

As an illustration of the traversal, consider traversing the structure in Figure 6-1(a). Figure 6-5 illustrates it. First, the field `root` of $T_0$ is traversed, then the field `left` of $N_0$ is traversed and finally the field `right` of $N_0$ is traversed. Therefore, we visit the nodes in the order $[N_0, N_1, N_2]$, which is the desired order. Thus this structure is among those that are generated. When we perform the traversal on the structure in Figure 6-1(b), we visit the nodes in the order $[N_1, N_0, N_2]$, which is not the desired order and therefore this structure is not generated.

We add the following fact to break symmetries:

```
module bst

sig BinarySearchTree {
  root: option Node,
  size: Int
}

sig Node {
  left: option Node,
  right: option Node,
  info: Int
}

fun BinarySearchTree::ClassInvariant() { ... }

fun Main() {
  all b: BinarySearchTree | b..ClassInvariant()
}

run Main for 3 but 1 BinarySearchTree, 2 Int
```

Figure 6-4: Alloy specification for a binary search tree.

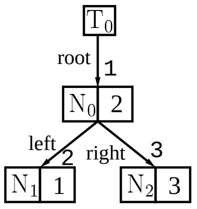

Figure 6-5: Traversing the nodes of a binary search tree. Fields are labeled additionally with a number that indicates the order in which a particular field is traversed. For clarity we do not show the field `size` of $T_0$.

```
fact BreakSymmetries {
  // if tree is non-empty, root is the first node in linear order
  BinarySearchTree.root in OrdFirst(Node)

  all n: BinarySearchTree.root.*(left + right) {
    // define traversal and require nodes to be visited in order
    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))
  }
}
```

The addition of this fact to our binary search tree specification rules out generation of the structure illustrated in Figure 6-1(b) but not of the structure illustrates in Figure 6-1(a).

#### Default Values for Unreachable Nodes

For the component of a `BinarySearchTree` instance that is not reachable from root, we do not care about the field values and therefore we express the following fact to assign pre-defined fixed values to fields of unreachable nodes:

```
fact Unreachable {
  all n: Node - BinarySearchTree.root.*(left + right) {
    // fields have pre-defined fixed values
    n.info = 0
    no n.left
    no n.right
  }
}
```

### 6.4.4 Implementing Traversals for Graphs with an Acyclic Backbone

The above example inspires a specific technique, which we next present, for writing the traversal constraints for a whole class of structures, in particular for rooted structures that have nodes of the same type and that have an *acyclic backbone*. Of course, traversals for this class and for other classes of structures can be written in several other ways.

A structure $S$ rooted at $s$ has an acyclic backbone if there exists a set $F$ of fields such that all nodes in the structure are reachable from $s$ via 0 or more traversals of fields in $F$ and $S$ has no undirected cycles that traverse only the fields in $F$. For example, red-black trees have an acyclic backbone: $F$ consists of all fields except `parent`. Doubly-linked circular lists, however, do not have an acyclic backbone.

A traversal on structure $S$ rooted at $s$ with an acyclic backbone along the set of fields $F = \{f_1, \ldots, f_n\}$ may be defined in two steps as follows. One, express a formula for breaking symmetries on the reachable component:

- linearly order the nodes
- $s$ is the first node in order
- for all reachable nodes $n$, for all $i$, either $n.f_i$ is `null` or $n.f_i$ is the node that comes immediately after the largest node that is reachable from $n$ along a traversal that first traverses a field $f_j$ where $j < i$ and then traverses fields in $F$ zero or more times.

Two, express a formula for setting default values for any unreachable component.

## 6.5 Summary

The user can efficiently break all symmetries by: (1) defining a total order on atoms and fields; (2) defining a traversal from the root that linearizes the structure; and (3) requiring that elements not reachable from the root have given default values. This process forces generation to produce the canonical structure for each isomorphism class, but it requires the user to manually add symmetry-breaking predicates.

Defining traversals using a set of visited nodes in an imperative notation is straightforward. However, in a declarative notation, such as Alloy, defining traversals can be more involved. This is because for structures that do not have acyclic backbones, traversals need to keep explicit track of visited nodes so that the ordering constraints are enforced only when a node is first visited. Nonetheless, for common data structures, it is straightforward.

Our traversal technique is valid for generating nonisomorphic structures that can be represented using rooted directed graphs where an edge connects a node to exactly one node. Allowing an edge to connect a node to more than one nodes compromises the optimality.

# Chapter 7

# Case Studies

This chapter presents case studies that evaluate (1) the feasibility of using SAT solvers in solving complex structural constraints and generating nonisomorphic structures (including some from the Java Collection Framework); and (2) the use of TestEra in checking stand-alone applications including a Java implementation of an intentional naming scheme for mobile networks and a C++ implementation of a fault-tree analysis system developed for NASA. The chapter concludes with a summary of the lessons learned and key aspects of our studies.

The studies with stand-alone applications enabled us to identify some useful modeling patterns. For example, in both the studies the Alloy model of data structures abstracts details of the structures in the implementation and the (manually written) concretization translation generates appropriate inputs.

We use the mChaff [71] solver in the experiments. It is worth pointing out that even though the version of mChaff we use supports solution enumeration, it is not optimized for enumeration. Despite that, the experimental results indicate that it is still feasible to generate high quality test suites using off-the-shelf SAT technology. Naturally, as more efficient solvers become available, the performance of our test generation technique would improve accordingly.

We have published results on structure generation at SAT 2003 [54] and the Intentional Naming Scheme study at SoftMC 2001 [51].

## 7.1 Structure Generation Using SAT

Recall the two key computational steps involved in test generation: (1) using a SAT solver to solve structural constraints and (2) concretizing the solutions found by SAT into actual tests. For a variety of data structures, we report the results on using off-the-shelf SAT technology to perform the first step and on using our TestEra prototype to perform the second step. We present the results of mChaff's performance in solving structural constraints and compare its performance with that of relsat [7], another off-the-shelf solver that supports enumeration. To discount the time it takes to write solutions to a file, we slightly modified the standard distributions of mChaff and relsat to disable solution reporting so that the solvers, despite actually computing the solutions, report only the total number of solutions found for each formula. (In Section 7.1.4, we tabulate results for test generation and these results include the times for file I/O.) We have also conducted a preliminary investigation on using Binary Decision Diagrams in place of SAT and we present those results in this section also.

To evaluate the overall cost of test generation and correctness checking using our prototype, we report also the cost of performing abstraction translations and the cost of checking whether a method preserves the class invariant.

The experiments were performed on a 1.8 GHz Pentium 4 processor with 2 GB of RAM.

### 7.1.1 Benchmarks

Table 7.1 presents the results of mChaff for a set of benchmark formulas that represent structural invariants. Each benchmark is named after the class for which data structures are generated; the structures also contain objects from other classes.

`BinarySearchTree` is our running example. `LinkedList` is the implementation of linked lists in the Java Collections Framework, a part of the standard Java libraries. This implementation uses doubly-linked, circular lists that have a `size` field and a `header` node as a sentinel node [24]. (Linked lists also provide methods that allow them to be used as stacks and queues.) `TreeMap` implements the `Map` interface using red-black trees [24]. Each node has a `key` and a `value`. `HashSet` implements the `Set` interface, backed by a hash table [24]; this implementation builds collision lists for buckets with the same hash code. `HeapArray` is an array-based implementation of a heap (or an implementation of a priority queue) [24].

### 7.1.2 Example benchmark

To give a flavor of the Java benchmarks we use, we describe the red-black tree [24] benchmark as implemented in the Java Collection Framework. (Appendix B describes the properties of red-black trees and how we express them in Alloy.)

The class `java.util.TreeMap` implements red-black trees. Part of the `TreeMap` class declaration is:

```
class TreeMap {
    Entry root;
    ...
    static class Entry {
        Object key;
        Entry left;
        Entry right;
        Entry parent;
        boolean color;
        ...
    }
}
```

|  |  |  | manual symmetry breaking |  |  |  | automatic symmetry breaking |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|
| benchmark | size | #prim | #vars | #clauses | #sols | time | #vars | #clauses | #sols | time |
| `BinarySearchTree` | 6 | 86 | 2120 | 6686 | 132 | 1.05 | 2333 | 7018 | 357 | 1.50 |
|  | 7 | 114 | 3165 | 10375 | 429 | 6.46 | 3439 | 10786 | 1866 | 7.45 |
|  | 8 | 146 | 4504 | 15216 | 1430 | 40.46 | 4831 | 15682 | 10286 | 64.40 |
|  | 9 | 182 | 7775 | 29618 | 4862 | 548.69 | 8141 | 30103 | 60616 | 1049.93 |
| `LinkedList` | 6 | 146 | 2017 | 6597 | 203 | 0.38 | 2520 | 7419 | 5975 | 3.46 |
|  | 7 | 191 | 2834 | 9834 | 877 | 1.04 | 3559 | 11021 | 52392 | 68.71 |
|  | 8 | 242 | 3837 | 14007 | 4140 | 4.76 | 4432 | 14939 | 734296 | 4637.99 |
|  | 9 | 299 | 5852 | 24411 | 21147 | 36.52 | 6629 | 25630 | — | — |
| `TreeMap` | 6 | 203 | 5203 | 15162 | 20 | 9.10 | 5542 | 15668 | 322 | 10.85 |
|  | 7 | 263 | 7578 | 22095 | 35 | 110.42 | 8076 | 22842 | 1160 | 69.09 |
|  | 8 | 331 | 10578 | 30896 | 64 | 254.13 | 11265 | 31930 | 4185 | 583.62 |
|  | 9 | 407 | 16111 | 51115 | 122 | 741.55 | 17017 | 52482 | 16180 | 3873.99 |
| `HashSet` | 6 | 285 | 5254 | 19079 | 462 | 6.06 | 5798 | 19865 | 693 | 7.04 |
|  | 7 | 373 | 7540 | 28881 | 1716 | 31.52 | 8270 | 29918 | 3172 | 30.04 |
|  | 8 | 473 | 10392 | 41430 | 6435 | 151.42 | 11102 | 42342 | 15011 | 167.30 |
|  | 9 | 585 | 15380 | 63308 | 24310 | 511.51 | 16277 | 64441 | 73519 | 1587.72 |
| `HeapArray` | 5 | 56 | 544 | 1178 | 1919 | 0.55 |  |  |  |  |
|  | 6 | 72 | 704 | 1611 | 13139 | 5.10 |  |  |  |  |
|  | 7 | 90 | 884 | 2128 | 117562 | 62.62 |  |  |  |  |
|  | 8 | 110 | 1084 | 2735 | 1005075 | 1171.64 |  |  |  |  |

Table 7.1: Performance of mChaff. All times are in seconds (of total elapsed wall-clock time). For sizes larger than presented, enumeration of solutions for automatically constructed symmetry-breaking predicates takes longer than 1 hour.

`TreeMap` implements a mapping between keys and values and an `Entry` has two data fields: `key` and `value` (which is not shown above). The field `value` represents the value that the corresponding `key` is mapped to and does not constrain the structure of a red-black tree. `TreeMap` has some other fields that we have not presented above. Some of these fields are constants, e.g., the field `RED` is the constant boolean `false`, and some are not relevant for testing commonly used methods (such as `remove`, `get` and `put`), e.g., the field `modCount` is used to detect co-modification. For test generation, TestEra allows users to specify which fields to exclude from the Alloy models it builds.

The declared type of `key` is `Object`. However, `key` objects need to be compared with each other as red-black trees are binary search trees. For comparisons, either an explicit `Comparator` for keys can be provided at the time of creation of the tree or the natural ordering of the actual type of `key` objects can be used. We define the actual type of the field `key` to be `java.lang.Integer`. Recall that, TestEra allows users to assign to a field a type (different from the declared type) that is actually used in test generation.

### 7.1.3 SAT Performance

#### mChaff

Table 7.1 shows results for several input sizes for each benchmark. All bound parameters are set exactly to the given size; e.g., all lists have exactly the given number of nodes and the elements come from a set with the given size. For each size, we use mChaff to enumerate solutions for two CNF formulas:

- one with symmetry-breaking predicates generated fully automatically (using the default values of the Alloy Analyzer);

- one with symmetry-breaking predicates added entirely manually to Alloy specifications (as described in Chapter 6).

We tabulate the number of primary variables (i.e., the variables that directly represent the values of signatures and relations in the corresponding Alloy specification), the total number of variables (which includes the number of variables that are introduced as a result of translation to CNF), the number of clauses, the number of solutions, and the time it takes to generate all solutions. The time shows the total elapsed time from the start of mChaff's execution to the end of generation of all solutions (without writing them in a file). It is worth noting that the time to generate solutions often accounts for more than one-half of the time TestEra takes to test a benchmark data structure implementation; thus, improving the efficiency of solution enumeration would significantly improve TestEra's overall performance.

For `BinarySearchTree`, `LinkedList`, `TreeMap`, and `HashSet`, the numbers of nonisomorphic structures are given in the Sloane's On-Line Encyclopedia of Integer Sequences [81]. For all sizes, formulas with manually added symmetry-breaking predicates have as many solutions as the given number of structures, which shows that these predicates eliminate all symmetries. For `HeapArray`, no symmetry-breaking is required: two array-based heaps are isomorphic if and only if they are identical, since they consist only of integers (i.e., array indices and heap elements) that are not permutable. For testing, it is desirable to generate only nonisomorphic inputs since without breaking isomorphisms it would be impractical to systematically test on all inputs. The factor by which the total number of solutions (including isomorphic solutions) is more than the total number of nonisomorphic solutions is exponential in the input size. For example, for `TreeMap` and size nine, there are more than 44 million total solutions, whereas there are only 122 nonisomorphic solutions.

In all cases, formulas with automatic symmetry breaking (using default parameter values) have more solutions than formulas with manual symmetry breaking. Also, in most cases it takes longer to generate the solutions for formulas with automatic symmetry breaking; a simple reason for this is that enumerating more solutions usually takes more time. This is not always the case, however: for `HashSet` and `TreeMap` of size seven, it takes less time to enumerate more solutions. This illustrates the general trade-off in (automatic) symmetry breaking: adding more symmetry-breaking predicates can reduce the number of (isomorphic) solutions, but it makes the boolean formula larger, which can increase the enumeration time. Note that merely the number of variables and clauses does not dictate how many symmetries are broken. For example, in all examples but `HeapArray`, the manual approach generates fewer variables and clauses than the automatic approach, yet the manual approach breaks more symmetries. This is because a manual predicate rules out more isomorphic instances per literal of the predicate, so it is "denser" [79]. The Alloy Analyzer allows users to tune symmetry breaking; we have experimented with different parameter values and the analyzer's default values seem to achieve a sweet spot for our benchmarks.

Note that we do not present numbers for `LinkedList` of size nine with automatic symmetry breaking; for this formula mChaff runs out of memory (2 GB). This suggests that for enumeration, the scheme for clause learning in mChaff [71] may need

| benchmark | size | # sols | mChaff time | relsat time |
|---|---|---|---|---|
| `BinarySearchTree` | 6 | 132 | 1.05 | 4.81 |
| | 7 | 429 | 6.46 | 36.28 |
| | 8 | 1430 | 40.46 | 268.22 |
| `LinkedList` | 6 | 203 | 0.38 | 1.21 |
| | 7 | 877 | 1.04 | 9.08 |
| | 8 | 4140 | 4.76 | 78.40 |
| `TreeMap` | 6 | 20 | 9.10 | 19.22 |
| | 7 | 35 | 110.42 | 128.27 |
| | 8 | 64 | 254.13 | 665.50 |
| `HashSet` | 6 | 462 | 6.06 | 52.49 |
| | 7 | 1716 | 31.52 | 475.00 |
| | 8 | 6435 | 151.42 | 4100.99 |
| `HeapArray` | 5 | 1919 | 0.55 | 6.71 |
| | 6 | 13139 | 5.10 | 77.12 |
| | 7 | 117562 | 62.62 | 1073.49 |

Table 7.2: Performance comparison of mChaff with relsat in solution enumeration for benchmark formulas with manually added symmetry breaking predicates. All times are in seconds (of total elapsed wall-clock time).

to be modified to (dynamically) adjust according to the available memory perhaps at the expense of performance, when enumerating all solutions. If there is no effective pruning or simplification of clauses added in order to exclude the already found solutions, complete solution enumeration can become infeasible. For all other benchmark formulas, mChaff is able to enumerate all solutions, even when there are more than a million of them.

**Comparison of mChaff with Relsat**

Table 7.2 presents the performance comparison of mChaff with relsat in enumerating *all* solutions for benchmark formulas with manually added symmetry breaking constraints. Enumeration by mChaff is dramatically more efficient than that of relsat for the benchmark data structures. For these benchmarks, it happens that mChaff's default enumeration does not generate any solutions with "don't care" bits. However, we believe mChaff's enumeration technique of obtaining partial solutions with don't-care variables (such that any completion of the solution satisfies the CNF) would also be useful for complete enumeration as grounding out partial solutions with "don't care" bits takes time linear in the number of new solutions generated. Perhaps this technique would outperform relsat's technique of always producing complete solutions.

| benchmark | size | # structures | # gen per sec | # conc per sec | # abst per sec | # check per sec |
|---|---|---|---|---|---|---|
| `BinarySearchTree` | 6 | 731 | 111 | 637 | 416 | 34 |
| `LinkedList` | 6 | 19608 | 1392 | 1241 | 551 | 98 |
| `TreeMap` | 6 | 327 | 83 | 405 | 210 | 3 |
| `HashSet` | 6 | 3384 | 232 | 534 | 237 | 6 |
| `HeapArray` | 6 | 13139 | 2171 | 2431 | 411 | 91 |

Table 7.3: Test generation performance. Number of structures represents the number of all structures up to the tabulated size. The number of boolean solutions per second generated by mChaff, the number of Java tests concretized per second, the number of Java outputs abstracted per second and the number of input/output pairs checked for correctness are tabulated.

**Binary Decision Diagrams**

We also conducted some very preliminary experiments using Binary Decision Diagrams (BDDs) in place of SAT solvers. Intuitively, BDDs seem attractive because they make it easier to read off all solutions, once a BDD for a formula has been obtained. Of course, the construction of a BDD itself may be infeasible and can take a long time (and exponential space). We experimented with the CUDD [82] BDD package. We constructed BDDs bottom-up, using automatic variable reordering via sifting [14], from the boolean DAGs from which the CNFs were produced. For all benchmarks, the BDD approach scaled poorly; for nontrivial sizes (over five), the BDD construction led to unmanageably large BDDs (over a million nodes) and did not finish within the allotted time limit of 10 minutes. These results are preliminary and we believe BDD experts might be able to fine tune the performance of BDDs to provide efficient enumeration.

## 7.1.4 Test Generation

In test generation, the structures enumerated by a SAT solver are concretized into actual Java objects, on which methods can be invoked. This section presents results of generating actual Java objects using the automatic translations that TestEra provides. For structure enumeration, we use a version of mChaff that outputs solutions to CNF formulas to disk. We included, in the times for structure generation and concretization, the time to perform disk operations.

We also present results for abstracting Java outputs into Alloy instances and the results for correctness checking by using post-conditions as test oracles, where the post-condition simply requires preservation of the class invariant. Since the times for generation, concretization, abstraction, and checking are independent of the time it takes for the method under test to execute, we tabulate, for each benchmark, the results for testing the method that takes one input—the benchmark structure—and has an empty body.

Table 7.3 tabulates the experimental results, for structures up to size 6. A recent study [64] shows that structures no larger than this size are required to get full statement and branch coverage on a range of methods for the tabulated benchmarks.

### Concretization

As we would expect, concretization of an instance on average is faster than its actual generation in most cases. The only exception is the `LinkedList` benchmark, which is partly due to the fact that our translations as currently implemented are generic and can be significantly optimized through automatic generation of dedicated translations that target a specific benchmark for a specific scope, and partly due to the fact that the structural constraints of a doubly-linked list represent the simplest of the constraints among our benchmarks. Naturally, the time to generate a structure is influenced by the complexity of the structural constraints. On the other hand, the time to concretize is determined simply by the size of an instance.

The results show efficient enumeration even by a solver that is not (at present) optimized for enumeration. Notice that the overall test generation performance varies from a few dozen to a few hundred tests per second. This time compares favorably with the time it would take to generate suites manually even by experienced testers.

### Abstraction

For the tabulated benchmarks, abstraction translates a few hundred instances per second but is slower than concretization. Recall that abstraction builds a set of visited nodes to avoid getting into an infinite cycle, which is not required in concretization. Even though we use hash sets and therefore concretizations and abstractions should have asymptotically the same running time, the observed difference is due to differences in the constants involved in the running times and perhaps also due to the use of reflection in obtaining field names.

Paradoxically, the operation that actually evaluates boolean formulas for given solutions is a bottleneck in the overall performance of TestEra. The reason is that in the Alloy Analyzer, this operation is intended to be used for one instance at a time and not for checking thousands of instances, and is supported only as a secondary feature—the primary use of the analyzer, of course, being to *solve* constraints. This operation can be significantly optimized. Indeed, checking whether a formula evaluates to true on a given solution, theoretically speaking, is a much less expensive operation than finding a solution for the formula. It is also worth pointing out that testing can be performed even in the absence of checking given post-conditions. In particular, if desired, we can simply execute the method and check whether any execution raises a runtime exception, such as `NullPointerException` or `ArrayIndexOutOfBoundsException`. Of course, we can also simply check runtime assertions on automatically generated inputs.

## 7.2 Testing Stand-alone Applications

We next present experiments on using TestEra to test two stand-alone applications: (1) the Intentional Naming System (INS, implemented in Java) and (2) the Galileo fault-tree analysis tool (implemented in C++)[1]. In both systems, we discovered significant bugs that were not known to the system designers[2].

### 7.2.1 Intentional Naming System

The Intentional Naming System (INS) [3] is a naming architecture (implemented in Java) for resource discovery and service location in dynamic networks. We focused on testing the core naming infrastructure of the system and in particular on the `Lookup-Name` algorithm that allows clients to locate services of interest. We tested some key partial correctness properties of the algorithm, which include published claims.

For test generation, we built an Alloy model of the key data structures of INS and for correctness checking, we modeled in Alloy the relevant correctness criteria. TestEra identified significant flaws in the INS implementation. These flaws also existed in the INS design [50], and we first corrected the design. We then modified the implementation of INS and checked its correctness using TestEra. (See [51] for details of the study.)

The original Java implementation of the core naming architecture of INS [78] consists of around 2000 lines of code.

#### Intentional Names

Services, in INS, are referred to by *intentional names*, which describe properties that services provide, rather than by their network locations. An intentional name is a tree consisting of alternating levels of *attributes* and *values*. The `Query` in Figure 7-1(a) is an example intentional name; hollow circles represent attributes and filled circles represent values. The query describes a camera service in building NE-43. A wildcard may be used in place of a value to show that any value is acceptable.

Name resolvers in INS maintain a database that stores a mapping between service descriptions and physical network locations. Client applications invoke a resolver's `Lookup-Name` method to locate services of interest. Figure 7-1(a) illustrates an example of invoking `Lookup-Name`. `Database` stores description of two services: service `R0` provides a camera service in NE-43, and service `R1` provides a printer service in NE-43. Invoking `Lookup-Name` on `Query` and `Database` should return `R0`.

#### Methodology

We performed this study using the Alloy 1.0 tool-set. In particular, the input constraints and test oracle were written in the Alloy 1.0 language [44]. We re-used the constraints we had written previously in our analysis [50] of the design of INS. These

[1]The Galileo case-study is joint work with David Coppit of College of William and Mary, and Kevin Sullivan and Jinlin Yang of the University of Virginia.

[2]For the INS, we had also previously discovered these bugs [50] at the design level using a hand-built Alloy model of the data structures and key algorithms of INS; we reused that model of data structures in this study.

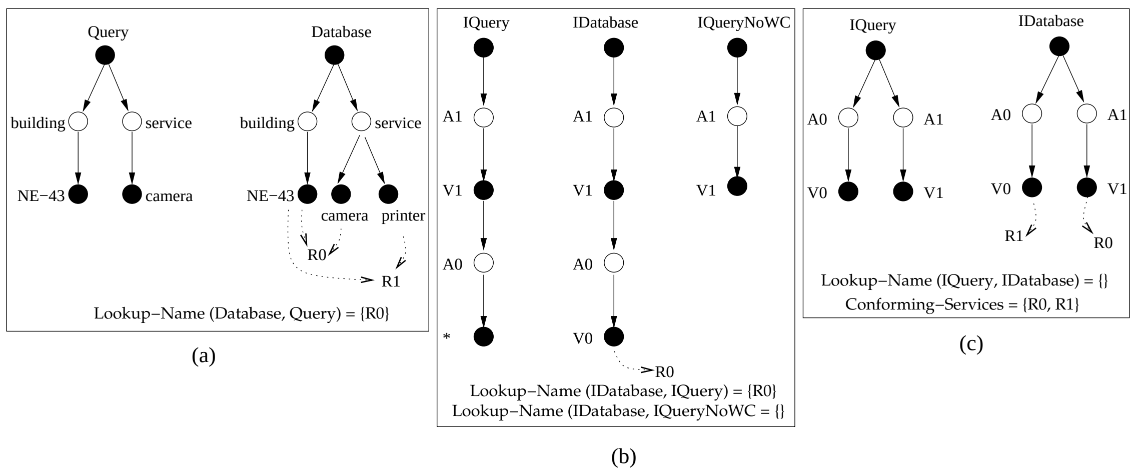
Query
Database
building
service
NE−43
camera
printer
R0
R1
Lookup−Name (Database, Query) = {R0}
(a)
IQuery
IDatabase
IQueryNoWC
A1
V1
A0
*
V0
R0
Lookup−Name (IDatabase, IQuery) = {R0}
Lookup−Name (IDatabase, IQueryNoWC = {}
(b)
IQuery
IDatabase
A0
A1
V0
V1
R1
R0
Lookup−Name (IQuery, IDatabase) = {}
Conforming−Services = {R0, R1}
(c)

Figure 7-1: (a) Intentional names in INS. `Query` describes a camera service in building `NE-43`. `Database` stores descriptions of two services: service `R0` provides a camera service in `NE-43`, and service `R1` provides a printer service in `NE-43`. Invoking `Lookup-Name` on `Query` and `Database` should return `R0`. (b) TestEra's counterexample to wildcard claim. `IQueryNoWC` is the same as `IQuery`, except for the omission of the wildcarded attribute `A0`. Different results of the two invocations of `Lookup-Name` contradict the claim. (c) TestEra's counterexample to monotonicity of addition. Both services `R0` and `R1` are considered conforming to `IQuery` by the semantics of INS, but their co-existence in `IDatabase` makes both of them erroneously non-conforming to `IQuery`.

constraints were not expressed at the concrete representation level but instead at the abstract level [3]. For example, in a database at the abstract level, an attribute has a corresponding *set* of values, whereas in the implementation the set is implemented using a vector (or simply an array). Abstracting the implementation details allowed us (1) to write the structural constraints in a more intuitive fashion and (2) to generate smaller boolean formulas with fewer variables, which provided efficient enumeration. However, doing so necessitated writing the concretization and abstraction translations manually. We found these translations straightforward to write because it was simple to define an isomorphism between structures represented by the Alloy specification and those represented by the Java implementation. It took us about 8 hours to write the Java code that performed the translations.

To illustrate the nature of translations that we wrote by hand, consider once again the aforementioned property that an attribute in a database can have several children values. In the Java implementation of INS, each attribute has a `children` field of type `java.util.Vector`. We model this property in Alloy as a relation from attributes to values. To concretize, we systematically translate tuples of the relation by adding elements to the `children` field of appropriate attribute objects. Similarly, to abstract we systematically iterate over the elements of `children` and add tuples to the relation.

#### Properties Checked and Results

Our checking of INS using TestEra focuses on the `Lookup-Name` method. `Lookup-Name` returns the set of services from the input database that *conform* to the input query. To investigate the correctness of `Lookup-Name`, we test its soundness (i.e., if it returns only conforming services) and completeness (i.e., if it returns all conforming services). The INS inventors did not state a formal definition of conformance, but only certain properties of `Lookup-Name`.

The published description of `Lookup-Name` claims: "This algorithm uses the assumption that omitted attributes correspond to wildcards; this is true for both the queries and advertisements." TestEra disproves this claim; Figure 7-1(b) illustrates a counterexample. `IQueryNoWC` is the same as `IQuery`, except that the wildcarded attribute `A0` is removed. Different results of the two invocations of `Lookup-Name` contradict the claim.

TestEra also shows that addition in INS is not monotonic, i.e., adding a new service to a database can cause existing services to erroneously become non-conforming. Figure 7-1(c) illustrates such a scenario: both services `R0` and `R1` are considered conforming to `IQuery` by the semantics of INS, but their co-existence in `IDatabase` makes both of them erroneously non-conforming to `IQuery`. In other words, if we consider resolving `IQuery` in a database that consists *only* of advertisement by `R1`, `Lookup-Name` returns `R1` as a valid service; however, in `IDatabase` which includes *both* advertisements by `R1` and `R2`, the algorithm returns the empty-set. This flaw points out that INS did not have a consistent notion of conformance. Both preceding flaws exist in the original design and implementation of INS.

To correct INS, we first defined what it means for a service to conform to the client's query. A service $s$ conforms to a query $q$ if $q$ is a subtree of the name of $s$, where the wildcard matches any value. This means that a service is conforming to $q$ if it provides all the attributes and (non-wildcard) values mentioned in $q$ in the right order. TestEra's analysis of the original implementation of `Lookup-Name` with respect to this definition of conformance reported several counterexamples. We modified the implementation and re-evaluated the correctness of `Lookup-Name` using TestEra. This time TestEra reported no flaws, increasing the confidence that our changes have corrected the problems with INS. The corrected algorithm now forms a part of the INS code base.

Table 7.4 summarizes the results. It illustrates the effectiveness of systematic testing in finding subtle errors even using small input bounds.

### 7.2.2 Galileo

Galileo [22] is a fault-tree analyzer developed (in C++) for NASA. We used TestEra to generate input fault-trees for Galileo. To check correctness, we ran another fault-tree analyzer, NOVA [21], on the same inputs and compared the outputs of Galileo and NOVA for each input. NOVA is a more recently developed fault-tree analysis tool that has been based on a formal Z specification of fault-trees with the aim of using the tool as a test oracle. If the output of Galileo differed from NOVA, we knew there was

|  |  | *phase 1* |  | *phase 2* |  |
|---|---|---|---|---|---|
| *method/property tested* | *size* | *# test* | *gen [sec]* | *# pass* | *check [sec]* |
| `published claim` | 3 | 12 | 9 | 10 | 6 |
| `addition monotonic` | 4 | 160 | 14 | 150 | 9 |
| `Lookup-Name (original)` | 3 | 16 | 8 | 10 | 6 |
| `Lookup-Name (corrected)` | 3 | 16 | 0 | 16 | 6 |

Table 7.4: TestEra's analysis of the Intentional Naming System. All times are in seconds. A generation time of 0 sec indicates previously generated tests were re-used.

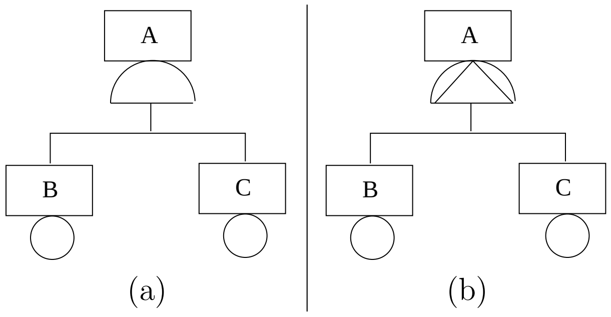

Figure 7-2: (a) Static fault tree; system level failure is an AND gate: if both events B and C occur, the system fails. (b) Dynamic fault tree; system level failure is a priority AND gate: if first event B occurs and then event C occurs, the system fails.

a flaw and we investigated further whether the bug was in Galileo or in NOVA or the specification itself was incorrect. TestEra found 20 distinct bugs that were not previously known. The section summarizes our methodology and tabulates performance results and presents an illustrative counterexample generated by TestEra.

### Fault Trees

Fault trees [88] model system failures; a fault tree represents the overall failure of a system as a combination of failures of basic components of the system. A *static* fault tree models how boolean combinations of component-level failure events produce system failures. Every fault tree has a *top-level* event that represents system level failure (and is graphically drawn as the root of the tree). Figure 7-2 (a) illustrates a static fault tree. The interior nodes of a tree are boolean *gates* and the leaves are *basic events*. The failure of basic events is characterized probabilistically by *basic event model* that consists of:

- the *rate parameter* (*lambda*) that defines the exponential distribution that characterizes the basic event's failure;
- the *coverage model* that defines the probabilities that the component masks

an internal failure (*res*), that a component fails in a way that can be detected by the system (*cov*) and that the component fails and brings down the system (*sing*); the sum of these three probabilities is 1; and

- the *dormancy value* (*dorm*) that defines the probability of failure if the basic event is used as an input to a gate to replace one of its failed inputs.

A basic event may have a *replication value* (*repl*), which allows the event to represent identical events connected to the same location(s) in a fault tree.

*Dynamic* fault trees augment static fault trees with constructs that allow modeling fault-tolerant systems; these constructs allow modeling, for example, how a sequence of events causes failure, and functional dependencies, such as failure of a trigger event causes failure of all dependent events. Figure 7-2 (b) illustrates a dynamic fault tree.

Fault trees can be represented graphically or with a *fault tree grammar* [21]. For example, the tree in Figure 7-2 (b) together with its basic event model can be textually represented as:

$$\begin{array}{l} \mathit{toplevel}\;\; A; \\ A\;\; \mathit{pand}\;\; B\;\; C; \\ B\;\; \mathit{lambda} = .01\;\; \mathit{cov} = .75\;\; \mathit{res} = 0\;\; \mathit{repl} = 1\;\; \mathit{dorm} = .25; \\ C\;\; \mathit{lambda} = .05\;\; \mathit{cov} = .5\;\; \mathit{res} = 0.1\;\; \mathit{repl} = 1\;\; \mathit{dorm} = .5; \end{array}$$

A *fault tree analyzer* computes, for a given fault tree and its failure models, the probability of system-level failure.

### Methodology

We used TestEra to generate test inputs in fault tree grammar. We based our Alloy model of fault trees on the Z specification. For input generation, we manually wrote a concretization translation (in Java) from Alloy instances to input strings.

Even for a very small bound there are a very large number of fault trees. For a bound of 4, there are in the order of a hundred million fault trees—part of this huge number is due to the choice in probabilistic models of failure. This necessitates generation to be restricted to exactly nonisomorphic trees to enable feasible enumeration. We manually add simple symmetry breaking predicates (as explained in Chapter 6) to enable such generation.

Despite symmetry breaking, the current version of mChaff was unable to enumerate all trees for a bound of 3 or more and ran out of (1 GB of) memory. To reduce the total number of inputs, we imposed additional constraints based on test purposes. In particular, we added constraints that ruled out generation of instances that uncovered previously known bugs. For example, we disallowed generation of trees that have both one or more priority-AND gate(s), and that have at least one priority-AND gate with input a basic event that has non unary replication.

A novel technique we employ in this study is to model in Alloy *partial* input trees that abstract away components that can be computed directly without the need

| bound | time (sec) to generate | # generated |
|---|---|---|
| 3 | 20291 | 1276324 |
| 4 | 322180 | 9529400 |

Table 7.5: Performance of TestEra in test generation for Galileo.

*toplevel* $Event_0$;
$Event_0$ $2of2$ $Event_1$;
$Event_1$ $lambda = .01$ $cov = .5$ $res = 0$ $repl = 2$ $dorm = .5$;

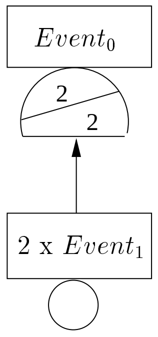

Figure 7-3: An input fault tree generated by TestEra. This input tree witnesses a bug in (the dynamic solver of) Galileo. The tree is a *kofm* gate (with $k$=$m$=2): the gate triggers if at least $k$ out of its $m$ inputs trigger. The basic event $Event_1$ has a replication value of 2 and the rate parameter, the coverage model and the dormancy value as given above.

for constraint solving. This helps the underlying SAT solver in enumerating the desired structures efficiently. Given an instance of the partial model, the concretization translation computes the remaining *derived* components (such as, whether the input denotes exactly one connected component) to generate a complete fault-tree.

### Results

Using a test suite with inputs within a bound of 4, we discovered bugs in not only NOVA and Galileo, but also the specification of fault trees. Table 7.5 tabulates a summary of the results. TestEra generates 1860 fault-trees per minute on average, which provides efficient enumeration. Using all generated inputs using the bound of 4, we discovered 20 bugs. It is worth pointing out that several of these bugs indicate significant semantic errors.

Figure 7-3 illustrates an input that exposes a bug in (the dynamic solver of) Galileo. For this input, the outputs of Galileo and NOVA differ. Galileo's *gate evaluation function*, which for an input gate determines whether any of its inputs have failed or whether they are all functional, behaves incorrectly on certain input gates. The original implementation of this function has an `if-then-else` block in which the code in the `if` block is, for correct behavior, not to be executed in mutual exclusion with the code in the `else` block.

## 7.3 Lessons

The following summarize the key lessons and aspects of the studies we have presented in this chapter:

- Systematic testing is feasible for real systems. Even though an input constraint is required for test generation, such constraints, even for real systems, often are simple to formulate. Further, correctness specifications can be as partial (or complete) as desired.

- Systematic testing can identify subtle bugs that have gone unnoticed despite years of use.

- Even though the current versions of SAT solvers are not optimized for enumeration, it is feasible to use them for enumerating complex structures. In some cases, however, mChaff crashed after generating an order of a million structures. This points to the need for further advancements in enumeration technology.

- Symmetry breaking plays a crucial role in enumeration. Writing symmetry breaking constraints for common data structures is simple but can become more involved for arbitrary structures.

- In modeling structures, it is often useful to identify *derived* components of a structure that can be computed directly (without the need for constraint *solving*) given values of the other components. Structure generation then proceeds in two steps. First, a constraint solver solves the desired constraints and second, an explicitly written program assigns appropriate values to the derived components.

- The ability to incorporate test purposes in input generation is a key requirement for developing tools for testing real systems.

# Chapter 8

# Discussion

This chapter discusses some limitations of the TestEra framework and our current implementation. We also address some feasibility issues of a systematic black-box testing approach based on first-order logic.

## 8.1 Unit Testing

TestEra is a framework designed primarily for unit testing of sequential Java programs. The notion of a unit, however, is not limited to one method. A sequence of method invocations can be tested identically to how a single method is tested. The unit can even represent an entire system as illustrated in the case-study for testing the command-line-interface of the Galileo fault-tree analyzer. However, TestEra cannot currently be used for system-level testing in a scenario where the user may interact with the system, for example a system that has a GUI front-end.

## 8.2 Unhandled Language Constructs

### 8.2.1 Primitive Types

To allow users to build specifications that include primitive types, we need to provide Alloy models for operations that Alloy does not support by default. Note that the use of SAT solvers as the underlying analysis technology necessitates a non-trivial treatment of such operations. For example, to support the addition operation for integers, we need to explicitly build a formula that encodes all valid triples (within the given scope) that represent the result of adding two integers.

Our current implementation provides limited support for (non-negative) `integer` and `boolean` types, including library code that automatically generates formulas for common integer operations, given an input scope. Support for (non-negative) integers is now also provided by default in Alloy itself.

We envision enabling the TestEra framework to use (in conjunction with SAT solvers) specialized decision procedures for handling operations on a variety of primitive types. One way to do so is to partition the constraints that describe an input according to whether they constrain fields of a primitive type or of a reference type, and to use a SAT solver for solving the structural constraints on reference fields, and appropriate decision procedures for solving constraints on primitive data. Since we a priori impose bounds on generated inputs, solving constraints is decidable simply by virtue of having a bounded space of candidate inputs.

### 8.2.2 Exceptions, Arrays, Inheritance etc.

Even though our current implementation does not support checking exceptional behavior of programs and generating inputs with multi-dimensional-array-based components, doing so is straightforward in principle. For example, to check whether a program correctly raises a certain exception on particular inputs, we could introduce a signature to model the exception and modify the abstraction algorithm accordingly. Programs that can raise exceptions or that use array-based structures internally can, nonetheless, be tested for their normal behavior.

Also, we do not currently support automatic initialization of field values (e.g., for `final` fields), which would require building an Alloy model corresponding to the Java code that does the initialization.

In object-oriented programs, inheritance plays a fundamental role. So far we have not addressed how to utilize class hierarchy in test generation. We have proposed a systematic treatment of inheritance in a first-order setting elsewhere [67]; we have not yet implemented this approach.

### 8.2.3 Multi-threadedness

Dynamically checking the correctness of multi-threaded programs for deadlocks and race conditions requires the ability to control thread scheduling. We envision using a model checker in conjunction with a SAT solver (and perhaps other decision procedures) to check for multi-threaded programs that operate on complex structures (similar to [55]).

## 8.3 Ease of Specification

Even though the use of expressions that represent heap traversals using operators, such as transitive closure, is intuitive and allows building succinct formulas, the use of quantifiers in building a specification can provide a learning challenge for developers who are not adept at programming in a declarative style.

A key issue with building specifications that define complex structures is to correctly write constraints that precisely define the desired structure. For example, it may be easy to recognize structures that are and are not trees, but hard to characterize them formally. There are various strategies we can use to enhance our confidence in the specification and to detect whether the specification is under- or over-constrained: 1) the Alloy tool-set allows users to visualize instances that satisfy given constraints; users can scroll through different instances to inspect for violation of an expected property; 2) users can formulate the same constraint as different formulas and use the Alloy Analyzer to check whether these formulas are equivalent; 3) for several common data structures, the number of nonisomorphic structures for various benchmark data structures (including red-black trees) and sizes appears in the Sloane's On-Line Encyclopedia of Integer Sequences [81]; for these structures, users can simply compare the number of structures enumerated by the analyzer with the published number; 4) users can employ existing libraries of common data structures. We used these strategies for building specifications for the presented studies; we built each specification within a day's work.

A recently proposed approach [64] to building specifications is to directly support common specification patterns, such as acyclicity, in the specification language. These patterns not only allow users to easily specify common data structure properties, but also provide more efficient generation of structures. In the context of TestEra, we would like to investigate how we can guide the SAT solver's search to exploit these patterns; as a first step, we would like to define a *pattern-aware* translation from first-order logic to boolean logic. We would also like to explore identifying such patterns automatically from a given an input constraint—quantified formulas built using path expressions seem to provide a notation that is amenable to such synthesis[1].

## 8.4 Systematic Black-box Testing

TestEra performs black-box [10] (or specification-based) testing. In other words, when testing a method the implementation of the method is not exploited in the generation of tests.

### 8.4.1 Test Generation from Input Constraints

The input constraints that TestEra uses for test generation can include constraints derived (manually, or automatically, if possible) from an implementation to support white-box testing but we do not present in this work a treatment of white-box testing.

As noted before, the input constraints can be manually strengthened to include test purposes, say to generate more useful instances. For example, we can rule-out generation of inputs that witness previously identified bugs—this also allows us to identify new bugs without having to fix the ones previously discovered.

### 8.4.2 Testing on All Small Nonisomorphic Inputs

TestEra's approach to testing is to systematically test the program on all nonisomorphic inputs within a small input size. A clear limitation of this approach is that it fails to explore program behaviors that are witnessed by large inputs only. For example, to test a program that sorts an input array using different sorting algorithms depending on the size of the array, it would be natural to test on inputs within a range of sizes to check each of the algorithms (at least) on a few inputs. A strategy TestEra users can

[1]Observation contributed by Daniel Weise.

apply is to test systematically on all small inputs and test selectively on a few larger inputs. Clearly, this strategy can be guided by the structural coverage of the program to give users more confidence about the correctness of their programs. Users can also gain confidence by iteratively increasing the bound and testing on larger inputs as permitted by time constraints.

### 8.4.3 Program Coverage on Small Inputs

A recent application [64] of the Korat testing framework [13, 64] evaluates how program coverage varies with bound on the input size for implementations of several benchmark data structures. (See Section 9 for more details on Korat.) The experiments show that it is feasible to achieve full statement and branch coverage for several benchmarks by testing on all inputs within a small input size.

### 8.4.4 Combining Small and Large Inputs

Even though these experiments indicate that systematic testing on all inputs is useful for testing data structure implementations, for certain kinds of programs, e.g., programs that manipulate arrays or programs that employ different algorithms for inputs of different sizes, certain kind of errors, e.g., array index out of bounds errors, can only be found on inputs that are sufficiently large. In some cases, it may be possible to scale down the constants used (for resource bounds) in a program and increase the effectiveness of testing on small inputs. However, this requires the ability to analyze an implementation and even exploiting the implementation details may be hard if the constant values have (implicit) dependencies. An approach that combines suites containing all small tests with suites containing some (randomly sampled) larger tests seems likely to provide a sweet spot for finding bugs.

## 8.5 Application of SAT

We have developed an unconventional application of SAT solvers, for software testing. Our application requires a solver that can enumerate all satisfying assignments; each assignment provides a (non-isomorphic) input for the program. In this context, symmetry breaking plays a significant, but different role from its conventional one: rather than reducing the time to finding the first solution, it reduces the number of solutions generated, and improves the quality of the suite of test inputs.

We envision various other applications of solution enumeration. One natural application is in checking certain classes of logic formulas. For example, consider the formula $\forall x \in D.\ P(x) \Rightarrow Q(x)$, where $D$ is some (finite) domain, and $P$ and $Q$ are arbitrary predicates. We simply use a solver to enumerate all $x$ that satisfy $P(x)$ and then, for each such $x$, check that $Q(x)$ holds. Alternatively, we check the validity of the implication (without requiring solution enumeration) by using a solver to directly check satisfiability of the negation: $P(x) \wedge \neg Q(x)$. Usually, the latter approach is preferred because it "opens" $Q$ for the sophisticated optimizations that SAT solvers perform. However, when $Q$ is a very large formula (or a formula that cannot be easily constructed explicitly) and when $P$ is very constraining, the approach with solution enumeration may work better.

A desirable feature for solvers that can enumerate solutions is to allow users to control the order of enumeration. For example, for testing databases, we would like to get "similar" test cases one after the other so that we can restore the state by using "deltas" and built-in support for rollback, instead of always building the state from scratch. For checking programs, it is desirable to have a solver generate all solutions in the neighborhood (as defined by a given metric) of a particular solution; this would enable testing, for example the entire neighborhood of an execution that gets "close" to a bug.

As input size increases, the number of inputs typically grows quite significantly. If for a certain desired input size, there simply is too huge a number of inputs, one could test on a random sample of them by using a solver that supports enumeration in a random order. Alternatively, to test for such a size, one could first define a stronger notion of isomorphism, which takes into account the domain of application or even the implementation code, and then enumerate inputs (which are now potentially fewer in number than before). For example, consider generating all trees with at most three nodes that contain integer elements. Let us assume that it is only the structure of the tree that is relevant with respect to the code being tested. Then for a tree with two nodes it does not matter whether the actual elements are 1 and 2 or they are 1 and 3 and what matters is the order in which they appear in the tree.

Indeed, various test purposes can be incorporated to strengthen constraints and to reduce the number of possible solutions. Thus even though it may seem that solvers that support enumeration would always suffer due to the requirement of having to explicitly generate solutions, developing solvers that provide efficient enumeration is still useful since the number of solutions can generally by reduced to a manageable size.

# Chapter 9

# Related Work

There is a large body of research on specification-based testing. We are not aware, however, of any work prior to TestEra that allowed systematic generation of structurally complex tests from constraints that define the structures at the representation level.

In this chapter, we first discuss how TestEra relates to other projects on specification-based testing. Next, we compare TestEra with static analysis (and in particular model checking of software); although TestEra performs testing, i.e., dynamic analysis, it does so systematically for all inputs within a given scope, which makes it more like some static analyses.

## 9.1 Specification-based Testing

Using constraints to represent inputs is not a new idea and dates back at least three decades [19, 42, 56, 74]; the idea has been implemented in various tools including EFFIGY [56], TEGTGEN [57], and INKA [35]. But the focus of prior work has been to solve constraints on primitive data, such as integers and booleans, and not to solve constraints on complex structures, which requires very different constraint solving techniques. Some of the more recently developed frameworks do support generation of complex structures. Most notably, Korat [13, 64] provides an algorithm for nonisomorphic generation of complex structures from constraints given as imperative predicates. (We describe Korat in more detail below.)

An early paper by Goodenough and Gerhart [34] emphasizes the importance of specification-based testing. Various projects automate test case generation from specifications, such as Z specifications [30, 41, 83, 85], UML statecharts [73, 75], ADL specifications [15, 77], or AsmL specifications [36, 37]. These specifications typically do not involve structurally complex inputs, such as linked data structures illustrated in TestEra's case studies. Further, these do not address advanced programming constructs, such as polymorphic types.

The first version of AsmLT Test Generator [36] was based on finite-state machines (FSMs): an AsmL [37] specification is transformed into an FSM, and different traversals of FSM are used to construct test inputs. Korat adds structure generation to generation based on finite-state machines [36]. AsmLT was successfully used for detecting faults in a production-quality XPath compiler [84].

Dick and Faivre [29] were among the first to use an FSM-based approach: their tool first transforms a VDM [48] specification into a disjunctive normal form and then applies partition analysis to build an FSM. This work influenced the design of tools such as CASTING [4] and BZTT [60]. Unlike TestEra, these tools readily handle sequences of method calls. But these tools cannot handle structurally complex inputs.

There are many tools that produce test inputs from a description of tests. QuickCheck [17] is a tool for testing Haskell programs. It requires the tester to write Haskell functions that can produce valid test inputs; executions of such functions with different random seeds produce different test inputs. TestEra differs in that it requires only an invariant that characterizes valid test inputs and then uses a general-purpose search to generate *all* valid inputs. DGL [68] and lava [80] generate test inputs from production grammars. They were used mostly for random testing, although they can also systematically generate test inputs. However, they cannot easily represent structures with complex invariants. Even though DGL is Turing-complete and in theory it is possible to specify complex structures, doing so for a structure would essentially be the same as (and require as much effort as) writing a dedicated generator for that particular structure.

AETG [20] is a popular system for generating test inputs that cover all pair-wise, or $k$-wise (where $k$ is less than or equal to the number of parameters) combinations of test parameters. These parameters correspond to object fields in TestEra. Using pair-wise testing is applicable when parameters are relatively independent. However, in object-oriented programs object fields are dependent. This dependence allows SAT solvers to prune their search, which enables TestEra to provides efficient enumeration of inputs for such programs. Additionally, TestEra takes into account isomorphism and generates only one input from each isomorphism partition. In theory, AETG can generate structures by exhaustively checking all possible structures for validity. However, such an approach is infeasible in practice.

Cheon and Leavens developed jmlunit [16] for testing Java programs. They use the Java Modeling Language (JML) [59] for specifications; jmlunit automatically translates JML specifications into test oracles for JUnit [9]. This approach automates execution and checking of methods. However, the burden of test case generation is still on the tester who has to provide sets of possibilities for all method parameters and construct complex data structures using a sequence of method calls.

### 9.1.1 Korat

The Korat [13,64,65] framework uses JML specifications and automates both test case generation and correctness checking. Input constraints are written as Java predicates. For a given predicate and a *finitization* (that bounds the predicate's input space), Korat uses a novel search algorithm to systematically explore the predicate's input space and generate all nonisomorphic inputs (within the given finitization bound) for which the predicate returns true. The heart of Korat is to monitor executions of the predicate on candidate inputs and to prune the search based on the fields accessed during executions. The key observation behind Korat's pruning is that a (deterministic) predicate[1] that returns false without reading certain fields of its input would return false for any values of those fields. A conventional SAT solver prunes its search in a similar fashion: if a CNF clause evaluates to false and the clause contains only a subset of the boolean variables in the formula, the whole formula is false irrespective of any assignment of values of the variables that do not appear in the clause.

Korat has primarily been used for black-box testing [13], but it can be also used for white-box testing [64]. In black-box testing, the predicate given to Korat typically represents the method precondition, and thus inputs that satisfy the predicate represent valid inputs for the method under test. In white-box testing, the predicate represents, in addition to the precondition, the method body and the (negation of the) method postcondition, and thus it "opens up" the method body, which assists in searching for counterexamples in a goal-directed fashion. One way to similarly "open up" the method body in a TestEra-like approach is to translate the body into a first-order logic formula (akin to [89]). Such a translation provides a static way of checking for bugs but generates very large boolean formulas and it is not clear how well it scales to testing real systems.

TestEra and Korat can be primarily compared in two aspects: ease of specification and performance of testing. There is no clear winner in any aspect so far, and we view TestEra and Korat as complementary approaches. For beginner users of TestEra and Korat, the users familiar with Java find it easier to write specifications in JML (for Korat) than in Alloy (for TestEra)–this is not surprising, because JML specifications are based on familiar Java expressions–whereas the users familiar with Alloy typically find it easier to write Alloy specifications that also tend to be more succinct than their JML equivalents. Regarding the performance of testing, the main factor is the performance of test input generation. Generation in Korat is more sensitive to the actual way a specification is written: for any two equivalent specifications (where the TestEra specification includes manually added symmetry breaking predicates), TestEra takes about the same time to generate inputs. But with specifications written to suit Korat, it generates inputs faster than TestEra. Korat is amenable to the use of dedicated generators that make the generation even faster, while making the specifications easier to write. It is worth noting that Korat does not require any symmetry breaking to be provided by the user. Korat's generation of nonisomorphic structures is fully automatic and provably correct [64]. Having said that, a specification written not to suit Korat would make generation in Korat slower than in TestEra.

## 9.2 Static Analysis

Several projects aim at developing static analyses for verifying program properties. The Extended Static Checker (ESC) [28] uses a theorem prover to verify partial cor-

[1]Korat does not require predicates to be deterministic.

rectness of classes annotated with JML specifications. ESC has been used to verify absence of errors such as null pointer dereferences, array bounds violations, and division by zero. However, ESC cannot verify complex properties of linked data structures, such as invariants of red-black trees, because the decision procedures in the underlying theorem prover are not powerful enough to prove the formulas that arise in verification of code that operates on red-black trees. There are some recent research projects that attempt to address this issue. The Three-Valued-Logic Analyzer (TVLA) [62, 76] is the first static analysis system that can verify preservation of list structure in programs that perform list reversals via destructive updates to the input list. TVLA has been used to analyze very small programs that manipulate doubly-linked lists and circular lists, as well as some sorting programs. The pointer assertion logic engine (PALE) [70] can verify a large class of data structures that can be represented by a spanning tree backbone, with possibly additional pointers. These data structures include doubly-linked lists, trees with parent pointers, and threaded trees. While TVLA and PALE are primarily intraprocedural, Role Analysis [58] supports compositional interprocedural analysis and verifies similar properties.

While static analysis of program properties is a promising approach for ensuring program correctness in the long run, the current static analysis techniques can only verify limited program properties. For example, none of the above techniques can verify correctness of implementations of balanced trees, such as red-black trees. Testing, on the other hand, is very general and can verify any decidable program property for realistically large implementations, but for inputs bounded by a given size.

### 9.2.1 Checking by Translating Code to SAT Formulas

Conceptually, TestEra checks that the code under test satisfies the formula $\forall i \in I.\ pre(i) \Rightarrow (\forall o \in O.\ code(i, o) \Rightarrow post(i, o))$, where *pre* is precondition, $I$ is input domain, $O$ is output domain, $code(i, o)$ denotes execution on input $i$ that results in output $o$, and *post* is postcondition.

It is possible in some cases to translate (Java) code into a formula *code* and look for a counter-example using a SAT solver (see e.g. [12, 61, 90]). A nice property of such an approach is that it allows using any SAT solver, whereas TestEra requires a SAT solver that can enumerate all solutions. However, translation-based approaches typically build a formula, namely $pre \wedge code \Rightarrow \neg post$, which is much bigger than the formula that TestEra builds, namely *pre*. Therefore, TestEra works better for larger code that does not have many inputs, whereas translation-based approaches work better for smaller code that has many possible inputs.

The translation-based approach of Vaziri and Jackson allows analyzing Java methods that manipulate linked data structures [47, 89, 90]. This approach translates a bounded computation sequence into an Alloy model and checks the model exhaustively with the Alloy Analyzer. Although static, this approach requires specifications for all helper methods and is unsound with respect to both the size of input and the length of computation. Since TestEra checks actual executions, it does not require specifications for helper methods, and it checks entire computations for larger inputs.

## 9.3 Software Model Checking

There has been a lot of recent interest in applying model checking to software. JavaPathFinder [91] and VeriSoft [32] operate directly on a program and systematically explore its state space to check correctness. Bandera [23] and JCAT [27], translate Java programs into the input language of existing model checkers, such as SPIN [40] and SMV [69]. They handle a significant portion of Java, including dynamic allocation, object references, exceptions, inheritance, and threads. They also provide automated support for reducing a program's state space through program slicing and data abstraction. SLAM [5, 6] uses predicate abstraction and model checking to analyze C programs for correct calls to API.

Most of the work on applying model checking to software has focused on checking event sequences, specified in temporal logic or as "API usage rules" in the form of finite state machines. These approaches offer strong guarantees: if a program is successfully checked, there is no input/execution that would lead to an error. However, they typically did not consider linked data structures or considered them only to reduce the state space to be explored and not to check the data structures themselves. TestEra, on the other hand, checks correctness of methods that manipulate linked data structures, but provides guarantees only for the inputs within the given bound.

A recent backtracking-based technique [55] combines a novel algorithm for solving (on-the-fly) structural constraints with traditional symbolic execution [56], and enables systematic testing of multi-threaded programs with linked structures. To our knowledge, this is the first technique that shows how to enable an off-the-shelf model checker to check simultaneously for properties of structures and properties of thread interleavings. An implementation of this technique in the Java PathFinder model checker has been used to systematically generate inputs for flight control software [39].

# Chapter 10

# Conclusion

This chapter points out the importance of testing, summarizes our work on automating testing of modern (object-oriented) software and concludes the thesis with some final thoughts.

## 10.1 Importance of Testing

Software systems are steadily growing in complexity and size. At the same time, reliability is becoming a more vital concern. To meet this demand for reliability, a great deal of progress is needed in building software checking techniques that developers would be willing to adopt.

Testing, the most widely used technique for validating software quality, is a labor-intensive process. Inadequate testing is blamed for $60 billion in annual costs incurred by users and vendors, according to a study conducted last year by the Commerce Department's National Institute of Standards and Technology. The study estimates that an improved testing infrastructure might save a third of these costs.

## 10.2 Constraint-based Test Generation

The key idea in this dissertation is systematic generation of structurally complex test data from declarative constraints that characterize the tests. For testing a method in an object-oriented program, these constraints represent the class invariants of the method inputs. But these constraints can additionally represent test purposes and test selection criteria. The ability to generate tests from constraints thus provides a powerful technique for testing real systems.

The TestEra tool implements this technique for testing Java programs. TestEra uses a SAT solver for test generation and correctness checking. Given a method precondition (that characterizes allowed inputs) and (optionally) a postcondition, and a bound on input size, TestEra (1) uses SAT to solve the constraint represented by the precondition, (2) translates each boolean solution into a concrete test input, (3) executes the method on each input, (4) translates each output into a boolean solution of the constraint represented by the postcondition, and (5) uses SAT to check if each input/output pair satisfies the postcondition.

We have performed various experiments to evaluate (1) the feasibility of using SAT solvers for solving constraints that represent structurally complex data and (2) the practicability of using systematic testing for checking the correctness of real applications. The experiments with generating complex data structures, including some that are implemented in the Java Collection Framework, indicate that it is feasible to use off-the-shelf SAT solvers for enumerating structures. The experiments with testing real applications uncovered significant flaws (that had gone unnoticed despite years of use) in an intentional naming scheme developed for dynamic networks and a fault-tree analyzer developed for NASA.

The use of SAT solvers for test generation presents a compelling application that suggests that solution enumeration in SAT is an important feature that merits research in its own right.

## 10.3 Final Thoughts

Writing constraints that express crucial properties of code, even before the code has been implemented, has great benefits. The success of test-first programming [8], a key practice of Extreme Programming, indicates not just the importance of having regression tests and having them early, but also the value of focusing on intent before coding. Even though test-first programming does not require the developer to write a specification, the act of devising test cases forces a consideration of what behavior is intended. And writing a constraint that defines a complex structure is often, surprisingly, much less work than building manually a reasonable test suite for a method that manipulates such a structure. This approach thus promises to bring the benefit of test-first programming at lower cost.

Writing a constraint formally nonetheless incurs a non-trivial cost. A key to enabling wide use of constraint-based testing is to develop techniques that allow building constraints without a huge overhead in terms of cost of writing. Such techniques need to provide well-founded (mathematical) notations that are accessible and cost-effective to non-experts, and to provide functionality that minimizes the burden of building specifications, for example by allowing developers to validate their specifications.

Among the notations that are currently in use, object models are becoming predominant, particularly for object-oriented design. They allow developers to reason about code at a more intuitive and abstract level. An object modeling notation, such as Alloy, that is based on a simple first-order logic of sets and relations, seems to provide a solid platform to build a notation that both has a mathematical foundation and is likely to gain wide acceptability among the developers. By writing an input constraint in Alloy, the developer can rely on the analyzer not only to generate a high quality set of test inputs but also to check crucial properties of the specification itself.

Our work on systematic testing provides, more generally, a technique for establishing a relationship between object models and code. Developing such techniques have various benefits. They not only allow establishing conformance of code to models but also allow developers to use a model as a surrogate for code and to have confidence that properties inferred from the model hold also for the code itself.

It is worth exploring how systematic testing might be extended for checking system level properties of industrial-scale applications that take arbitrary inputs. A fusion of black-box testing, white-box testing and symbolic-execution-based techniques, where input constraints are conjoined with constraints from code and solved using a combination of solvers, is worth investigating.

# Appendix A

# Implementation of `remove`

```
boolean remove(int i) {
    Node parent = null;
    Node current = root;
    while (current != null) {
        if (info == current.info) break;
        if (info < current.info) {
            parent = current;
            current = current.left;
        } else {
            // (info > current.info)
            parent = current;
            current = current.right;
        }
    }
    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;
}

// helper method
Node removeNode(Node current) {
    size--;
    Node left = current.left, 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;
}
```

# Appendix B

# Generating Red-black Trees

Red-black trees [24] are binary search trees with one extra bit of information per node: its *color*, which can be either “red” or “black”. By restricting the way nodes are colored on a path from the root to a leaf, red-black trees ensure that the tree is *balanced*, i.e., guarantee that basic dynamic set operations on a red-black tree take $O(\lg n)$ time in the worst case.

A binary search tree is a red-black tree if:

1. Every node is either red or black.
2. If a node is red, then both its children are black.
3. Every simple path from the root node to a descendant leaf contains the same number of black nodes.

The Java Collection Framework, in particular the class `java.util.TreeMap`, implements red-black trees. Part of the `TreeMap` class declaration is:

```
class TreeMap {
    Entry root;
    ...
    static class Entry {
        Object key;
        Entry left;
        Entry right;
        Entry parent;
        boolean color;
        ...
    }
}
```

`TreeMap` implements a mapping between keys and values and an `Entry` has two data fields: `key` and `value` (which is not shown above). The field `value` represents the value that the corresponding `key` is mapped to and does not constrain the structure of a red-black tree. `TreeMap` has some other fields that we have not presented above. Some of these fields are constants, e.g., the field `RED` is the constant boolean `false`, and

```
// binary search tree
// ...

// parent ok
all e, f: root.*(left+right) |
  e in f.(left + right) <=> f = e.parent

// red black tree
// 1. every node is red or black, by construction
// 2. red node has black children
all e: root.*(left + right) |
  e.color = false && some e.left + e.right =>
  (e.left + e.right).color = true
// 3. all paths from root to NIL have same # of black nodes
all e1, e2: root.*(left + right) |
  (no e1.left || no e1.right) && (no e2.left || no e2.right) =>
    #{p: root.*(left+right) | e1 in p.*(left+right) && p.color = true} =
    #{p: root.*(left+right) | e2 in p.*(left+right) && p.color = true}
```

Figure B-1: Structural constraints for red-black trees.

some are not relevant for testing commonly used methods (such as `remove`, `get` and `put`), e.g., the field `modCount` is used to detect co-modification. For test generation, TestEra allows users to specify which fields to exclude from the Alloy models it builds.

The declared type of `key` is `Object`. However, `key` objects need to be compared with each other as red-black trees are binary search trees. For comparisons, either an explicit `Comparator` for keys can be provided at the time of creation of the tree or the natural ordering of the actual type of `key` objects can be used. We define the actual type of the field `key` to be `java.lang.Integer`. Recall that, TestEra allows users to assign to a field a type (different from the declared type) that is actually used in test generation.

The structural constraints for red-black trees are readily expressible in Alloy (Figure B-1). The constraints require the tree to be a binary search tree, the `parent` field to be consistent with the fields `left` and `right`, and the tree to satisfy the properties of red-black trees given above.

The symmetry breaking constraints for red-black trees are identical to those for binary search trees (Section 6.4.3) since the field `color` has a type with values that are not permutable and the value of field `parent` is determined entirely by the values of `left` and `right` fields.

# Bibliography

[1] SAT competitions for comparing state-of-the-art solvers. `http://www.satlive.org/SATCompetition/`.

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

[3] 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.

[4] Lionel Van Aertryck, Marc Benveniste, and Daniel Le Metayer. CASTING: A formally based software test generation method. In *Proc. First IEEE International Conference on Formal Engineering Methods*, Hiroshima, Japan, November 1997.

[5] 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.

[6] Thomas Ball and Sriram K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc. 29th Annual ACM Symposium on the Principles of Programming Languages (POPL)*, pages 1–3, 2002.

[7] R. J. Bayardo Jr. and R. 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] Armin Biere. Limmat satisfiability solver. `http://www.inf.ethz.ch/personal/biere/projects/limmat/`.

[12] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Masahiro Fujita, and Yunshan Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proc. 36thConference on Design Automation (DAC)*, New Orleans, LA, 1999.

[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] Karl Brace, Richard Rudell, and Randal Bryant. Efficient implementation of a BDD package. In *Proc. of the Design Automation Conference (DAC)*, pages 40–45, 1990.

[15] 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.

[16] 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.

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

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

[19] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, September 1976.

[20] 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.

[21] David Coppit. *Engineering Modeling and Analysis: Sound Method and Effective Tools.* PhD thesis, The University of Viginia, Charlottesville, VA, 2003.

[22] David Coppit and Kevin J. Sullivan. Galileo: A tool built from mass-market applications. In *Proc. 22nd International Conference on Software Engineering (ICSE)*, Limerick, Ireland, June 2000.

[23] 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.

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

[25] J. Crawford, M. Ginsberg, E. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *Proc. Fifth International Conference on Principles of Knowledge Representation and Reasoning*, 1996.

[26] Markus Dahm. Byte code engineering library. `http://bcel.sourceforge.net/`.

[27] C. Demartini, R. Iosif, and R. Sisto. A deadlock detection tool for concurrent Java programs. *Software - Practice and Experience*, July 1999.

[28] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, 1998.

[29] Jeremy Dick and Alain Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *Proc. Formal Methods Europe (FME)*, pages 268–284, 1993.

[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] Joanne Bechta Dugan, Kevin J. Sullivan, and David Coppit. Developing a low-cost high-quality software tool for dynamic fault tree analysis. *Transactions on Reliability*, pages 49–59, 1999.

[32] 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.

[33] Evgueni Goldberg and Yakov Novikov. BerkMin: a fast and robust SAT-solver. In *Proceedings of Design, Automation, and Test in Europe (DATE)*, March 2002.

[34] J. Goodenough and S. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, June 1975.

[35] 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.

[36] 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.

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

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

[39] Mats P. E. Heimdahl, Sanjai Rayadurgam, Willem Visser, Devaraj George, and Jimin Gao. Auto-generating test sequences using model checkers: A case study. In *Proc. 3rd International Workshop on Formal Approaches to Testing of Software (FATES)*, Montreal, Canada, October 2003.

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

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

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

[43] Daniel Jackson. Micromodels of software: Modelling and analysis with Alloy, 2001. `http://sdg.lcs.mit.edu/alloy/book.pdf`.

[44] Daniel Jackson. Alloy: A lightweight object modeling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2), April 2002.

[45] Daniel Jackson and Alan Fekete. Lightweight analysis of object interactions. In *Proc. Fourth International Symposium on Theoretical Aspects of Computer Software*, Sendai, Japan, October 2001.

[46] 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.

[47] 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.

[48] Cliff B. Jones. *Systematic Software Development using VDM.* Prentice-Hall, 1990.

[49] Henry Kautz and Bart Selman. Planning as satisfiability. In *Proc. European Conference on Artificial Intelligence (ECAI)*, Vienna, Austria, August 1992.

[50] Sarfraz Khurshid and Daniel Jackson. Exploring the design of an intentional naming scheme with an automatic constraint analyzer. In *Proc. 15th IEEE International Conference on Automated Software Engineering (ASE)*, Grenoble, France, Sep 2000.

[51] 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.

[52] Sarfraz Khurshid and Darko Marinov. TestEra: Specification-based testing of Java programs using SAT. *Automated Software Engineering Journal*, 2004. (to appear).

[53] Sarfraz Khurshid, Darko Marinov, and Daniel Jackson. An analyzable annotation language. In *Proc. ACM SIGPLAN 2002 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Seattle, WA, Nov 2002.

[54] 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.

[55] 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.

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

[57] Bogdan Korel. Automated test data generation for programs with procedures. San Diego, CA, 1996.

[58] 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.

[59] 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. (last revision: Aug 2001).

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

[61] K. Rustan M. Leino. A SAT characterization of boolean-program correctness. In *Proc. 10th International SPIN Workshop on Model Checking of Software*, 2003.

[62] Tal Lev-Ami and Mooly Sagiv. TVLA: A system for implementing static analyses. In *Proc. Static Analysis Symposium*, Santa Barbara, CA, June 2000.

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

[64] Darko Marinov. *Testing Using a Solver for Imperative Constraints*. PhD thesis, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 2004. (to appear).

[65] 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.

[66] 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.

[67] Darko Marinov and Sarfraz Khurshid. VAlloy: Virtual functions meet a relational language. In *Proc. Formal Methods Europe (FME)*, Copenhagen, Denmark, July 2002.

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

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

[70] 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.

[71] 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.

[72] Alexander Nadel. Jerusat SAT solver. `http://www.geocities.com/alikn78/`.

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

[74] C. V. Ramamoorthy, Siu-Bun F. Ho, and W. T. Chen. On the automated generation of program test data. *IEEE Transactions on Software Engineering*, 2(4), 1976.

[75] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley Object Technology Series, 1998.

[76] 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.

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

[78] Elliot Schwartz. Design and implementation of intentional names. Master's thesis, MIT Laboratory for Computer Science, Cambridge, MA, June 1999.

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

[80] 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.

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

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

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

[84] Keith Stobie. Advanced modeling, model based test generation, and Abstract state machine Language AsmL. `http://www.sasqag.org/pastmeetings/asml.ppt`, 2003.

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

[86] Kevin Sullivan, David Coppit, Jinlin Yang, Sarfraz Khurshid, and Daniel Jackson. A case study of specification-based testing using Alloy. (In preparation.).

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

[88] United States Nuclear Regulatory Commission. *Fault Tree Handbook*, 1981. NUREG-0492.

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

[90] 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.

[91] 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.

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