TestEra: Specification-based Testing of Java Programs Using SAT

Sarfraz Khurshid and Darko Marinov
({khurshid,marinov}@lcs.mit.edu)
MIT Lab for Computer Science, 200 Technology Square, Cambridge, MA 02139

Abstract. TestEra is a framework for automated specification-based testing of Java programs. TestEra requires as input a Java method (in sourcecode or bytecode), a formal specification of the pre- and post-conditions of that method, and a bound that limits the size of the test cases to be generated. Using the method's pre-condition, TestEra automatically generates all nonisomorphic test inputs up to the given bound. It executes the method on each test input, and uses the method postcondition as an oracle to check the correctness of each output. Specifications are first-order logic formulae. As an enabling technology, TestEra uses the Alloy toolset, which provides an automatic SAT-based tool for analyzing first-order logic formulae. We have used TestEra to check several Java programs including an architecture for dynamic networks, the Alloy-alpha analyzer, a fault-tree analyzer, and methods from the Java Collection Framework.

1. Introduction

Manual software testing, in general, and test data generation, in particular, are labor-intensive processes. Automated testing can significantly reduce the cost of software development and maintenance [6]. This paper describes TestEra, a framework for automated specification-based testing of Java programs. TestEra requires as input a Java method (in sourcecode or bytecode), a formal specification of the pre- and post-conditions of that method, and a *bound* that limits the size of the test cases to be generated; a test input is within a bound of k if at most k objects of any given class appear in it. Using the method's pre-condition, TestEra automatically generates all nonisomorphic test inputs up to the given bound. TestEra executes the method on each test input, and uses the method postcondition as a test oracle to check the correctness of each output.

Specifications are first-order logic formulae. As an enabling technology, TestEra uses the Alloy toolset. Alloy [25] is a first-order declarative language based on sets and relations. The Alloy Analyzer [27] is a fully automatic tool that finds *instances* of Alloy specifications: an instance assigns values to the sets and relations in the specification such that all formulae in the specification evaluate to true.

© 2004 *Kluwer Academic Publishers. Printed in the Netherlands.*

The key idea behind TestEra is to automate testing of Java programs, requiring only that the structural invariants of inputs and the correctness criteria for the methods be formally specified, but not the body of the methods themselves. This is in contrast to previous work [30] on analyzing a naming architecture, we modeled both inputs and computation in Alloy. We discovered that manually modeling imperative computation is complicated due to Alloy's declarative nature and lack of support for recursion. Automatically modeling computation in Alloy was performed [28] for a subset of Java. This approach has been optimized [53], but it does not seem to scale to checking programs say of the size of the aforementioned naming architecture.

Given the method precondition in Alloy, TestEra uses the Alloy Analyzer to generate all nonisomorphic [49] instances that satisfy the precondition specification. TestEra automatically *concretizes* these instances to create Java objects, which form the test inputs for the method under test. TestEra executes the method on each input and automatically *abstracts* each output to an instance of the postcondition specification. TestEra uses the Alloy Analyzer to check if this instance satisfies the postcondition. If it does not, TestEra reports a concrete counterexample, i.e., an input/output pair that violates the correctness specification. TestEra can graphically display the counterexample, e.g., as a heap snapshot, using the visualization of the analyzer. Thus, TestEra automates a complete suite of testing activities—generation of tests, generation and execution of a test driver (which includes concretization and abstraction translations), and error reporting.

TestEra can be used to identify why a program fails. When TestEra finds a counterexample, it also reports which constraints in the postcondition are violated, e.g., that the output list of a sorting method is sorted and acyclic but not a permutation of the input list. TestEra can also be used to test a method on "interesting" inputs by describing in first-order logic desired properties of inputs—TestEra would then test the method only on inputs that satisfy these properties.

Since TestEra can automatically generate Java data structures from a description of the structural invariants, it is able to test code at the concrete data type level. For example, in order to test a method that performs deletion on balanced binary trees, the input tree can automatically be generated from its structural description, without the need to construct it using a sequence of method calls. This is especially useful since it can be hard to determine the particular sequence of element insertions in an empty tree that would produce a balanced tree in a desired configuration, especially if some deletions need to be interleaved with insertions to generate such a configuration.

In this article, we describe the core components and analysis architecture of TestEra. We also show various applications of our implementation of TestEra. We illustrate TestEra's capabilities by checking not only intricate library methods that manipulate complicated data structures, but also stand-alone applications like the Alloy Analyzer itself. TestEra was able to identify subtle bugs in a part of the Alloy-alpha Analyzer; these bugs have been fixed in the current publicly available release.

The rest of this article is organized as follows. Section 2 presents an example of testing a linked data structure and illustrates how programmers can use TestEra to test their programs. Section 3 describes the basics of the Alloy specification language and its automatic tool support. Section 4 describes how we model in first-order logic the mutable state of an object-oriented program. Section 5 describes the key algorithms of the TestEra framework. Section 6 describes some case studies that we have performed with TestEra. Section 7 describes our TestEra prototype and discusses its performance. We present a discussion (of TestEra's limitations) in Section 8, discuss related work in Section 9 and conclude in Section 10.

2. Example

This section presents an example of testing a linked data structure to illustrate how programmers can use TestEra to test their programs.

The following Java code declares a singly linked list:

```
class List {
    Node header;
    static class Node {
        int elem;
        Node next;
    }
}
```

Each object of the class `List` represents a singly linked list. The `header` field represents the first node of a (non-empty) list; if empty, `header` is `null`. Objects of the inner class `Node` represent nodes of the lists. The field `elem` contains the (primitive) integer data in a node. The field `next` points to the next node.

Let's assume the class `List` implements acyclic lists. The class invariant of `List` can be stated using the following Alloy formula:

```
// acyclicity
all n: header.*next | not n in n.^next
```

In Alloy, '`*`' denotes reflexive transitive closure, '`^`' denotes transitive closure, '`:`' and '`in`' denote membership (mathematically, the subset relation). The expression `header.*next` thus denotes the set of all nodes reachable from the `header` node of a list following 0 or more traversals along the `next` field; similarly, the expression `n.^next` denotes the set using 1 or more traversals. The quantifier '`all`' stands for universal quantification. The class invariant thus states that for all nodes that are reachable from the `header` node of a list, traversing from such a node along the `next` field any number of times does not lead back to the same node.

Consider the following declaration of a (recursive) method that destructively updates its input list (represented by implicit parameter `this`) to sort it:

```
void mergeSort() {
    ...
}
```

The precondition for `mergeSort` is simply this class invariant. The postcondition for `mergeSort` includes the class invariant but also expresses stronger properties, e.g., the input list in the post-state is sorted[1] and moreover a permutation of the input list in the pre-state. These properties can be expressed in Alloy as follows:

```
// sorted
all n: header.*next | some n.next implies n.elem <= n.next.elem

// output is permutation of the input
all i: Integer |
 #{ n: header.*next | n.elem = i } =
 #{ n: header`.*next` | n.elem` = i }
```

In Alloy, `implies` denotes logical implication, '`#`' denotes set cardinality, '`{...}`' denotes a set comprehension expression. The backtick character '`` ` ``' denotes field traversal in the pre-state[2]. All other field traversals are in the default state, which is pre-state for a pre-condition and post-state for a post-condition.

The formula to check that output is sorted states that all nodes reachable from the header are in order; the formula employs the form

[1] We use the comparison operator `<=` here for ease of exposition; in TestEra we write the comparison `x <= y` as `LE(x, y)`, where `LE` is a library function for comparing integers.

[2] Alloy does not have a built-in notion of state; to ease writing post-conditions, TestEra defines a veneer on Alloy, in which backticks denote pre-state.

`some S` (meaning that `S` is non-empty) to ensure that elements are compared to their successors only if they exist. The formula to check that output is a permutation of input states that each integer occurs the same number of times in the list in the pre-state and in the list in the post-state.

Given the Java bytecode (or sourcecode) for `List` and `List$Node`[3], the precondition (which in this case is just the class invariant for `List`)[4] 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. TestEra builds appropriate Alloy specifications to use the Alloy Analyzer for test generation and correctness checking. TestEra also builds a Java test driver.

As an illustration of TestEra's checking, consider erroneously reversing a comparison in the helper method that merges lists from `(m.elem <= n.elem)` to `(n.elem >= m.elem)`; this results in `mergeSort` sorting the lists in reverse order. Using a bound of three[5], TestEra detects violation of the correctness criterion and generates counterexamples, one of which is the following:

```
counterexample found:
  pre-state
    this: 0 -> 0 -> 1
  post-state
    this: 1 -> 0 -> 0
```

TestEra's analysis also tells us that the list referenced by `this` in the post-state is a permutation of the list referenced by `this` in the pre-state but is not sorted.

It is worth noting that most shape analysis techniques [38, 46, 43] are either unable to handle methods like `mergeSort` or require invariants to be given explicitly by the user for loops and method calls. We discuss this further in Section 9.

3. Alloy

We describe next the basics of the Alloy specification language and the Alloy Analyzer; details can be found in [25–27]. We also briefly discuss nonisomorphic generation.

[3] `List$Node` denotes the inner class `Node` of class `List`.

[4] In general, since precondition is a formula on all method inputs, it may consist of several class invariants.

[5] A bound of two would also be sufficient to reveal this bug as all lists are sorted in the reverse order.

Alloy is a strongly typed language that assumes a universe of atoms partitioned into subsets, each of which is associated with a basic type. An Alloy specification is a sequence of paragraphs that can be of two kinds: signatures, used for construction of new types, 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 included in the current one using `open` declarations.

We next introduce relevant parts of Alloy using the list example. This section focuses on the syntax and semantics of Alloy. Section 4 explains the connection between mutable state of an object-oriented program and the Alloy models that TestEra builds.

3.1. Signature paragraphs

A signature paragraph introduces a basic (or uninterpreted) type. For example,

```
sig State {}
```

introduces `State` as a set of atoms. A signature paragraph may also declare a subset. For example,

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

declares `Pre` to be a subset of `State`. The qualifier `static` specifies the declared signature to contain exactly one element; the qualifier `part` declares `Pre`, together with any other subsets (of `State`) that have a `part` qualifier, to partition `State`.

A signature declaration may include a collection of relations (that are called *fields*) in it along with the types of the fields and constraints on their values. For example,

```
sig List {
  header: Node ?-> State }

sig Node {
  elem: Integer !-> State,
  next: Node ?-> State }
```

introduces `List` and `Node` as uninterpreted types. The field declaration for `header` introduces a relation of type `List -> Node -> State` (for some sig `Node`). The marking '`?`' indicates that for each `List` atom `l`, `l.header` is a relation of type `Node -> State` such that for each `State` atom `s`, `l.header` maps *at most one* `Node` atom to `s`; similarly the

markings '`!`' and '`+`' indicate respectively *exactly one* and *at least one*. We explain the dot operator '`.`' below in Section 3.2.1.

3.2. Formula paragraphs

Formula paragraphs are formed from Alloy expressions.

3.2.1. *Relational expressions*

The value of any expression in Alloy is always a relation—that is a collection of tuples of atoms. Each element of such a tuple is atomic and belongs to some basic type. A relation may have any arity greater than one. Relations are typed. Sets can be viewed as unary relations.

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 `T1 -> ... -> Tm` and `q` has type `U1 -> ... -> Un` such that `Tm = U1` and `m + n > 2`, `p.q` is a relation of type `T1 -> ... -> T(m-1) -> U2 -> ... -> Un` such that for each tuple `(t1, ..., tm)` in `p` and each tuple `(u1, ..., un)` in `q` with `tm = u1`, `(t1, ..., t(m-1), u2, ..., un)` is a tuple in `p.q`. When `p` is a unary relation (i.e., a set) and `q` is a binary relation, `p.q` is functional image, and when both `p` and `q` are binary relations, `p.q` is standard composition; `p.q` can alternatively be written as `p::q`, but with higher precedence.

The unary operators `~` (transpose), `^` (transitive closure), and `*` (reflexive transitive closure) have their standard interpretation; transpose can be applied to arbitrary binary relations, and closures can only be applied to homogeneous binary relations, whose type is `T -> T`.

3.2.2. *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 op e` consisting of a variable `v`, a comparison operator `op`, 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). In a declaration, `part` specifies partition and `disj` specifies disjointness; they have their usual meaning.

The declaration

```
disj v1,v2,... : e
```

is equivalent to a declaration for each of the variables `v1,v2,...`, with an additional constraint that the relations denoted by the variables are disjoint (i.e., share no tuple). Therefore, the formula

```
all disj v1, v2: e | F
```

is equivalent to

```
all v1, v2: e | no v1 & v2 => F
```

The declaration `part` additionally makes the union of variables be `e`.

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

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

has `n` parameters: the implicit parameter `this` of type `T1` and `p2, ..., pn` of types `T2, ..., Tn` respectively. The return value of a function is referred using the keyword `result` and the type of the return value of `f` is `R`. A function may not have an explicitly declared return type; the use of implicit parameter `this` is also optional.

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 correctness needs to be checked, assuming the facts in the model.

3.3. Alloy Analyzer

The Alloy Analyzer [27] is an automatic tool for analyzing models created in Alloy. Given a formula and a *scope*—a bound on the number of atoms in the universe—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. Since first order logic is undecidable, the analyzer limits its analysis to a finite scope. The

analysis [27] is based on a translation to a boolean satisfaction problem, and gains its power by exploiting state-of-the-art SAT solvers.

The models of formulae are termed *instances*. The following valuations of sets and relations introduced earlier in this section represent two distinct instances:

```
Instance 1:
-----------
State = {S0}
Pre = {S0}

int = {0, 1, 2}
List = {L0}
Node = {N0, N1, N2}

header = {(L0, N0, S0)}
next = {(N0, N1, S0), (N1, N2, S0)}
elem = {(N0, 0, S0), (N1, 0, S0), (N2, 1, S0)}

Instance 2:
-----------
State = {S0}
Pre = {S0}

int = {0, 1, 2}
List = {L0}
Node = {N0, N1, N2}

header = {(L0, N1, S0)}
next = {(N0, N2, S0), (N1, N0, S0)}
elem = {(N0, 0, S0), (N1, 0, S0), (N2, 1, S0)}
```

We use the standard definition of graph isomorphism for (edge-labeled graphs) to define *isomorphic* instances: atoms that do not represent primitive values are permutable. For example, `Instance 1` and `Instance 2` are isomorphic. (They represent the example input list illustrated in the counterexample given in Section 2.)

The analyzer can enumerate all possible instances of an Alloy model. The analyzer adapts the symmetry-breaking predicates of Crawford et al. [12] to provide the functionality of reducing the total number of instances generated—the original boolean formula is conjugated with additional clauses in order to produce only a few instances from each isomorphism class [49]. The input parameters to the analyzer can be set such that the analyzer enumerates exactly nonisomorphic instances. However, the resulting formulae tend to grow very large, which slows

down enumeration so that it takes more time to enumerate fewer instances. We have recently shown how to manually construct Alloy formulae that completely break isomorphs and also provide efficient generation for a variety of benchmark data structures [33]—we follow this approach here.

4. State

Alloy does not have a built in notion of state mutation. We next describe how we model the state of an object-oriented program[6] in a logic of sets and relations.

The heap of an executing program is viewed as a labeled graph whose nodes represent objects and whose edges represent fields. The presence of an edge labeled `f` from node `o` to `o'` says that the `f` field of the object `o` points to the object `o'`. 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 and partition the set of edges according to the declared fields.

To model mutation, we simply associate a distinct graph with each state. In a specification there are only two states—the pre-state and the post-state. Mathematically, we treat fields as ternary relations, each of which maps an object to an object in a given state.

For the singly linked list example of Section 2, we model the Java class and field declarations as three sets (`List`, `Node`, `int`) and three relations (`header: List -> Node -> State, elem: Node -> int -> State, next: Node -> Node -> State`) respectively, where `State` is a set with two atoms `Pre` and `Post` representing respectively the pre- and the post-state.

We model the value `null` as empty-set. In particular, to say the value of field `f` in object `o` is non-`null`, we express the formula ``` ``some o.f'' ```; similarly for `null`, we express ``` ``no o.f'' ```.

Notice that there exists a (trivial) isomorphism between (concrete) state of a program and our (abstract) model of state. This allows us to define straightforward algorithms to perform translations between the abstract and concrete domains; we present the algorithms in Section 5.

In building specifications, it is worth keeping in view the semantic data model of Alloy, which deals only with sets and relations. We are currently working on making it more intuitive for Java developers to write specifications in a logic of sets and relations [32]. We discuss this further in Section 8.

[6] We provide a treatment for programs that manipulate only reference or primitive types.

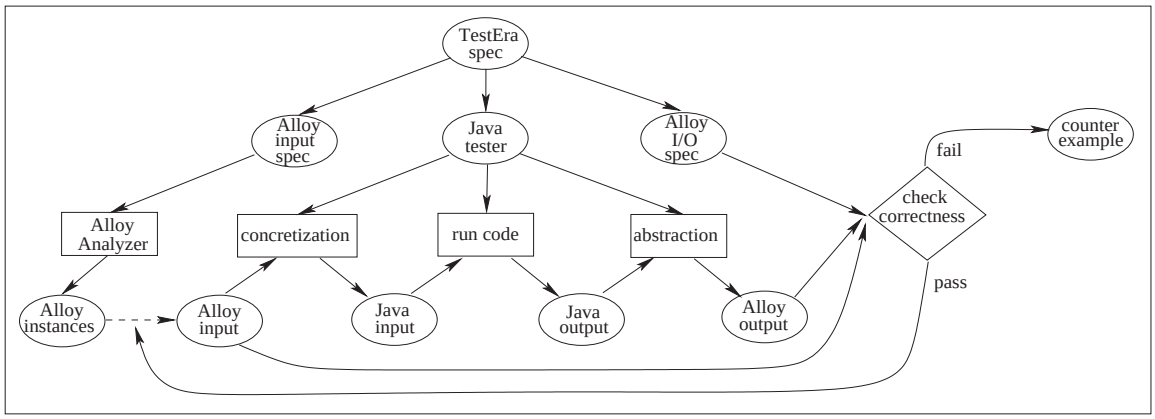

Figure 1. Basic TestEra framework

5. TestEra

TestEra is a novel framework for automated testing of Java programs. We have built TestEra upon Alloy and the Alloy Analyzer with the aim of checking actual Java implementations, without having to model Java computations in Alloy.

Figure 1 illustrates the main components of TestEra. A TestEra specification for a method states 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 our current implementation, we give this specification using command line arguments to the main TestEra method.

Given a TestEra specification, TestEra automatically 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 non-isomorphic instances of the Alloy input specification.

- In the second phase, each of the instances is tested 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. The output Alloy instance and the original Alloy input instance valuate the signatures and relations of the

```
AlloySpec generateInputSpec(Classfiles jar, Method m,
                            Formula pre) {
  result is alloy specification consisting of:
    sig declarations for (pre)state
    foreach class C in jar (which does not have a library spec)
      sig declaration for C
    foreach field f declared in class C
      if f is of reference type T
        field declaration, f: T ?-> State, in sig C
      if f is of primitive type T
        field declaration, f: T !-> State, in sig C
    function repOk that has signature corresponding to that of m
    and has as body pre (i.e., m's precondition), where
      each field f is replaced by f::Pre
}
```

Figure 2. Algorithm for creating input generation specification

Alloy input/output specification. TestEra uses the Alloy Analyzer to determine if this valuation satisfies the correctness specification. If the check fails, TestEra reports a counterexample. If the check succeeds, TestEra uses the next Alloy input instance for further testing.

5.1. Alloy specifications

We next discuss the Alloy specifications that TestEra automatically generates; these specifications are used by the Alloy Analyzer for generating inputs and checking correctness. Details of Alloy language can be found in [25] and of the Alloy Analyzer in [27]. See Section 3 for relevant details.

TestEra automatically parses Java classfiles and builds Alloy specifications. The current TestEra implementation uses the ByteCode Engineering Library [13] for bytecode parsing. TestEra builds Alloy specifications by combining signature declarations with Alloy functions that express given method pre and postconditions.

Alloy does not have a built in notion of state or mutation. The Alloy specifications that TestEra builds explicitly introduce state (following the approach we described in Section 4).

Figure 2 describes the basic algorithm TestEra uses to build the Alloy specification for input generation (the "Alloy input spec" in Figure 1). This algorithm first constructs appropriate sig and field declarations and next constructs an Alloy function (`repOk`) that represents the method precondition. Since precondition expresses a constraint on

```
AlloySpec generateOracleSpec(Method m, AlloySpec inputSpec,
                             Formula post) {
  result is inputSpec together with:
    sig declaration for post state
    function methodOk that has signature corresponding to m
    and has as body post (i.e., m's postcondition), where
      each backticked field f‘ is replaced by f::Pre
      each non-backticked field f is replaced f::Post
}
```

Figure 3. Algorithm for creating correctness checking specification.

the pre-state, the algorithm systematically introduces dereferencing in the pre-state. To generate input instances, TestEra uses the Alloy Analyzer to solve the constraints represented by `repOk` within the input scope[7].

Recall the singly linked list example introduced in Section 2. TestEra builds the following Alloy specification for generating inputs for the method `mergeSort`:

```
module mergeSort_phase_1

open java/primitive/integer

sig State {}
static part sig Pre extends State {}

sig List {
  header: Node ?-> State }

sig Node {
  elem: Integer !-> State,
  next: Node ?-> State }

fun List::RepOk() {
  // ayclic
  all n: header::Pre.*next::Pre | not n in n.^next::Pre }
```

Figure 3 describes the basic algorithm TestEra uses to build the Alloy specification for correctness checking (the "Alloy I/O spec" in Figure 1). The algorithm first constructs a sig declaration for post-state and next constructs an Alloy function (`methodOk`) that represents the method postcondition. Since a postcondition may express a constraint

[7] The scope of `State` for input generation is fixed at 1 since the pre-condition is a formula on the pre-state only and therefore there is only one state to consider.

that relates pre-state and post-state, the algorithm systematically introduces dereferencing in the appropriate state following the convention that backtick represents pre-state. To check correctness, TestEra uses the Alloy Analyzer to check whether the given input/output pair satisfies the constraints expressed in `methodOk` for the output scope[8].

For the singly linked list example (Section 2), TestEra builds the following Alloy specification for checking correctness of `mergeSort`:

```
module mergeSort_phase_2

open java/primitive/integer

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

sig List {
  header: Node ?-> State }

sig Node {
  elem: Integer !-> State,
  next: Node ?-> State }

fun List::RepOk() {
  // acyclic
  all n: header::Pre.*next::Pre | not n in n.^next::Pre }

fun List::methodOk() {
  // acyclicity
  all n: header::Post.*next::Post | not n in n.^next::Post

  // sorted
  all n: header::Post.*next::Post |
    some n.next::Post implies
    n.elem::Post <= n.next::Post.elem::Post

  // output is permutation of the input
  all i: Integer |
    # { n: header::Post.*next::Post | n.elem::Post = i } =
    # { n: header::Pre.*next::Pre | n.elem::Pre = i } }
```

[8] The scope of `State` for correctness checking is fixed at 2 since the post-condition is a formula that may relate pre-state and post-state, which are two distinct states.

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

  // for each atom create a corresponding Java object
  foreach (sig in a.sigs())
    foreach (atom in sig) {
      SigClass obj = new SigClass();
      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;
  for (i = 0; i < a.numParams(repOk); i++)
    inputs[i] = mapAJ.get(a.getParam(repOk, i));

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

Figure 4. Concretization algorithm `a2j`

5.2. Abstraction and concretization 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 appropriately executes the method to test. The translations are generated fully automatically when 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 manually provided.

A concretization, abbreviated `a2j`, translates Alloy instances to Java objects (or data structures). An abstraction, abbreviated `j2a`, translates Java data structures to Alloy instances.

Concretization `a2j` typically 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 4 describes a generic concretization algorithm `a2j`. The algorithm takes as input an Alloy instance and returns an array of Java

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

  while (!worklist.isEmpty()) {
    Object current = worklist.getFirst();
    if (!visited.add(current)) continue;
    foreach (field in getFields(current)) {
      Object to = getValue(field, current);
      if (to == null) continue;
      ret.addTuple(field, imap.mapJA.get(current),
                          imap.mapJA.get(to), Post);
      if (!visited.contains(to)) worklist.add(to);
    }
  }
}
```

Figure 5. Abstraction algorithm `j2a`

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[9]. In the second step, `a2j` sets values of objects according to tuples in the input relations; notice that all tuples represent values in pre-state. Finally, `a2j` builds an array of objects that represents a test input, e.g., 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.

Figure 5 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 `ret` by traversing the structures referenced by `result` and inputs in `imap` (in the post-state). This traversal is a simple worklist algorithm that tracks the set of visited objects. For each object that is visited for the first time, `j2a` adds tuples to `ret` according to field values of that object. For simplicity, we do not show the step in the algorithm

[9] For atoms that represent primitive values, TestEra uses library classes testera.primitive.*; for other atoms TestEra assumes accessibility permission for invoking appropriate constructors. TestEra also assumes accessibility permission for setting field values. Alternatively, TestEra could use Java's security manager mechanism to disable access checks for reflection.

that creates new atoms when it encounters an object that does not exist in the maps; this accounts for cases when a method allocates new objects that appear in the output but did not exist in the pre-state.

The algorithms `a2j` and `j2a` describe generic translations that make use of reflection. TestEra optimizes this by generating translations that are specialized for the particular method and relevant classes being tested, and do not use reflection.

TestEra generates Alloy specifications and abstraction and concretization translations automatically. The users may modify these specifications and translations, e.g., to introduce some abstractions in the Alloy specifications and improve efficiency of the analysis. However, introduction of abstraction in specifications requires manual generation of translations.

6. Case studies

We have used TestEra to check a variety of programs, including methods of some classes in the `java.util` package. Most of these programs manipulate non-trivial data structures. We have also tested a part of the Alloy Analyzer with TestEra. In this section, we illustrate some of the analyses performed by TestEra and the bugs that it detected.

6.1. Red-black trees (java.util.TreeMap)

We first outline TestEra's analysis of the red-black tree implementation given in `java.util.TreeMap` from the standard Java libraries (version 1.3).

Red-black trees [11] 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. Every leaf (NIL) is black.
3. If a node is red, then both its children are black.
4. Every path from the root node to a descendant leaf contains the same number of black nodes.

All four of these *red-black properties* are expressible in Alloy. We use TestEra to test the implementation of red-black trees given in `java.util.TreeMap`. In particular, we illustrate TestEra's analysis of the `remove` method in class `java.util.TreeMap`, which deletes a node with the given element from its input tree. Deletion is the most complex operation among the standard operations on red-black trees and involves rotations. The method `remove` in `java.util.TreeMap`, together with the auxiliary methods is about 300 lines of code.

Part of the `java.util.TreeMap` declaration is:

```
public class TreeMap {
    Entry root;
    ...
    static class Entry {
        Object key;
        Object value;
        Entry left;
        Entry right;
        Entry parent;
        boolean color;
        ...
    }
    public Object remove(Object key) {...}
    ...
}
```

Red-black trees in `java.util.TreeMap` implement a mapping between keys and values and therefore an `Entry` has two data fields: `key` and `value`. The field `value` represents the value that the corresponding `key` is mapped to. There are several fields of red-black trees that we have not presented above. Some of these fields are constants, e.g., field `RED` is the constant boolean `false` and some are not relevant for testing the `remove` method, e.g., `modCount`, which is used to raise `ConcurrentModificationException` and we do not currently check for exceptional behavior. TestEra allows users to specify which fields to exclude from the Alloy models it builds. We exclude from generation fields other than the ones declared above.

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, 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. TestEra allows users to define actual type of fields, which are then used for generation of objects. We define actual type of field `key` to be `java.lang.Integer`; TestEra automatically generates appropriate tree objects.

We use TestEra to generate red-black trees as test inputs for `remove` method. First, we need the class invariant for red-black trees:

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

// binary search tree properties
// unique keys
all disj e1, e2: root.*(left + right) |
  e1.key != e2.key
// distinct children
all e: root.*(left + right) |
  no e.(left+right) || e.left != e.right
// tree is acyclic
all e: root.*(left + right) |
  e !in e.^(left+right)

// left subtree has smaller keys
all e: root.*(left + right) |
  all el: e.left.*(left+right) |
    el.key <= e.key
// right subtree has larger keys
all e: root.*(left + right) |
  all er: e.right.*(left+right) |
    e.key <= er.key

// red black tree properties
// 1. every node is red or black, by construction
// 2. all leafs are black
// 3. red node has black children
all e: root.*(left + right) |
  e.color = false && some e.left + e.right =>
  (e.left + e.right).color = true
// 4. 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 }
```

The class invariant requires `parent` field to be consistent with the fields `left` and `right`, the tree to be a binary search tree and also to satisfy the four properties of red-black trees mentioned above.

After generating test inputs using the class invariant above, TestEra in phase 2 of its analysis tests `remove`. By default, TestEra checks the

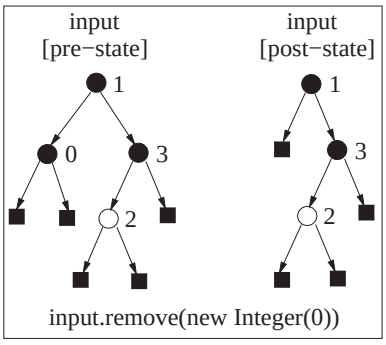

Figure 6. A counterexample for an erroneous version of `remove`. Nodes are labeled with the keys of the entries. Round filled nodes represent entries colored black and empty nodes represent entries colored red. Square filled nodes represent NIL nodes and are colored black. The entry with key `0` is to be deleted from the input red-black tree. The tree in post-state is not balanced; more precisely, property 4 is violated.

partial correctness property that the class invariant is preserved by the method. We can also check stronger properties, e.g., to check that the key to remove is actually removed from the tree, we can use the following post-condition:

```
root.*(left + right).key = root`.*(left` + right`).key - key
```

Recall, backtick (‘`’) indicates field access in pre-state.

As expected, TestEra's analysis of the original implementation provided in `java.util` does not produce any counterexamples. However, if we erroneously swap `BLACK` with `RED` in the following code fragment:

```
if (p.color == BLACK)
    fixAfterDeletion(p);
```

TestEra detects violation of structural constraints on red-black trees and produces concrete counterexamples. Figure 6 presents a counterexample. The tree in post-state is not balanced; more precisely, property 4 is violated.

It should be noted here that Alloy provides an expressive notation for writing properties of data structures. In contrast, the fourth property of red-black trees is not expressible in the PALE logic [43]. Similarly, TVLA [46] cannot check the `remove` method above.

6.2. Intentional Naming System

The Intentional Naming System (INS) [1] is a recently proposed naming architecture for resource discovery and service location in dynamic networks. In INS, services are referred to by *intentional* names, which

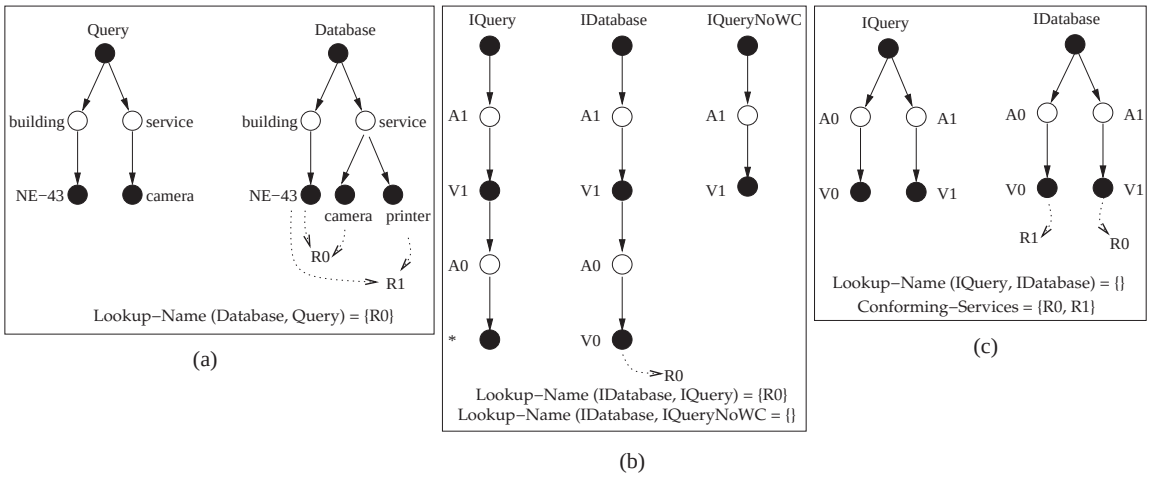

Figure 7. (a) Intentional names in INS, (b) and (c) counterexamples for `Lookup-Name`

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(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 resolver's `Lookup-Name` method to access services of interest. Figure 7(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`.

To illustrate the variety of TestEra's analyses, we discuss some flaws identified by TestEra in the Java implementation of INS [48]. These flaws actually existed in the INS design, and we first corrected the design. We then modified the implementation of INS and checked its correctness using TestEra. Details of our INS case study with TestEra can be found in [31]. The original Java implementation of INS [48] consists of around 2000 lines of code.

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 would like to test its soundness (i.e., if it returns only conforming services) and completeness (i.e., if it returns all conforming ser-

vices). However, the INS inventors did not state a definition of conformance and stated only certain properties of `Lookup-Name`. We investigate `Lookup-Name`'s correctness by checking whether it satisfies certain basic properties that are necessary for its correctness and a published claim about its behavior.

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(b) illustrates a counterexample: `IQueryNoWC` is the same as `IQuery`, except for the wildcarded attribute `A0`. 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(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 the notion of conformance between a service and a query: a service s is conforming 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 reports several counterexamples. We modified the implementation and re-evaluated the correctness of `Lookup-Name` using TestEra. This time TestEra reports no flaws, increasing the confidence that our changes have corrected the problems. The corrected algorithm now forms a part of the INS code base.

6.3. Alloy-alpha Analyzer

The main design goal for TestEra is that it efficiently analyzes complex data structures. But, TestEra can be applied also to test various other kinds of programs. As an illustration, we show how we used TestEra to uncover subtle bugs in the Alloy-alpha Analyzer.

The bugs appear in the analyzer because it generates instances that, for the user's convenience, retain the names that the user gives for

`static` signatures. The problems only appear in the rare case when the user explicitly declares a static subsignature with the same name as the one that the analyzer picks for an atom of a basic signature. These bugs have simple fixes and have now been corrected in the new version of the analyzer, which is publicly available for download.

Recall that the basic signatures in Alloy introduce new types. Therefore, distinct basic signatures must not share atoms, and the atoms within each signature must be also unique. We test the conjecture that instances produced by the old analyzer satisfy these properties.

We build an Alloy (meta-)specification of a simplified Alloy specification that consists only of basic signature and subsignature declarations. In phase 1, TestEra generates all non-isomorphic instances of this specification. Each of these instances I essentially represents an Alloy specification M. In phase 2, TestEra takes each instance I in turn and build the corresponding Alloy specification M. The testing next invokes the analyzer again to generate all instances of M and finally checks whether each such instance I' satisfies the uniqueness properties stated above.

The following Alloy code models an Alloy specification that consists only of signature declarations, with some of their atoms explicitly named (as `static` subsignatures).

```
sig SigName {}
sig Prefix {}
sig Suffix {}

sig Atom {
  namePrefix: Prefix,
  nameSuffix: Suffix }
fact AtomsHaveUniqueNames {
  all disj a1,a2:Atom |
    a1.namePrefix != a2.namePrefix ||
    a1.nameSuffix != a2.nameSuffix }

sig Sig {
  name: SigName,
  staticAtoms: set Atom }
fact SignaturesHaveUniqueNamesAndAtoms {
  all disj s1,s2:Sig |
    s1.name != s2.name &&
    no s1.staticAtoms & s2.staticAtoms}

static part sig Signature, Test extends SigName {}
static part sig S, T extends Prefix {}
static part sig Zero, One extends Suffix {}
```

Basic type `SigName` denotes signature names, and `Prefix` and `Suffix` build atom names. The fact `AtomsHaveUniqueNames` specifies that names of distinct atoms differ in either the prefix or the suffix. A `Sig` has a name and can have several atoms declared explicitly (i.e., its `static` subsignatures). The fact `SignaturesHaveUniqueNamesAndAtoms` constrains distinct signatures to have distinct names and atoms.

For the sake of simplicity, we let the pool for signature names be only {`Signature,Test`}, for prefixes {`S,T`}, and for suffixes {`Zero,One`}. (`Zero` and `One` are placeholders for symbols `0` and `1`, since these symbols without a leading alphabetic character are not allowed as subsignature names in Alloy.)

An example instance I that the analyzer generates for the above specification is:

```
SigName = {Signature, Test}
Prefix = {S, T}
Suffix = {Zero, One}
Atom = {A1, A2}
Sig = {Sig1, Sig2}

namePrefix={(A1, S), (A2, S)}
nameSuffix={(A1, One), (A2, Zero)}
name = {(Sig1, Test), (Sig2, Signature)}
staticAtoms = {(Sig1, A1), (Sig2, A2)}
```

This instance represents the Alloy specification M:

```
sig Test {}
sig Signature {}

static sig S1 extends Test {}
static sig S0 extends Signature {}
```

As stated earlier, for any instance generated by the analyzer, the valuations of signatures (and relations) in the instance must satisfy the uniqueness properties for the analyzer to be sound.

TestEra's analysis of this conjecture produces a counterexample. In particular, TestEra detects the following instance I' of M as produced by the analyzer:

```
Signature = {S0, S1}
Test = {S1}
```

This instance violates the property that atoms in distinct signatures must be distinct.

Table I. Summary of TestEra's analyses

case study	*method/property tested*	*size*	*phase 1*		*phase 2*	
			# test	*gen [sec]*	*# pass*	*check [sec]*
singly linked lists	`mergeSort`	3	27	9	27	7
	`mergeSort (erroneous)`	3	27	0	3	7
red black trees (`java.util`)	`remove`	5	70	26	70	19
	`remove (erroneous)`	5	70	0	50	18
INS	`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
Alloy-alpha	`disj sigs / unique atoms`	2	12	5	6	25

Another counterexample that TestEra generates is:

```
Signature = {S0,S0}
Test = {S1}
```

This instance also violates the property that atoms in a signature must be distinct. Both violations of the uniqueness properties also affect the visualization part of the analyzer. As mentioned, though, these bugs have been fixed in the new version of the Alloy Analyzer, which is publicly available for download.

7. Implementation and Performance

We have implemented TestEra in Java. Table I summarizes the performance of our implementation on the presented case studies; we conducted the analyses on a Pentium III, 700 MHz processor. We tabulate, for each case study, the method we test, a representative input size, and the phase 1 (i.e., input generation) and phase 2 (i.e., correctness checking) statistics of TestEra's checking for that size. For phase 1 we tabulate, the number of inputs generated and the time to generate these inputs. For phase 2, we tabulate the number of inputs that satisfy the correctness criteria and the total time for checking. A time of 0 seconds in phase 1 indicates reusing already generated tests. All times are in seconds. In all the cases, TestEra takes less than a minute to complete both the automatic generation of instances and the verification of correctness, for the given small input sizes.

In phase 1, TestEra typically generates several test inputs per second. Among the studies we have presented, the most complex structural invariants are those for red-black trees. Note that the number of possible states to consider for generating red-black trees with five nodes is over 2^{80}. Of course, the Alloy Analyzer prunes away most of these states, and that is why the test inputs are generated fairly quickly.

In phase 2, TestEra's performance depends on both the time to perform abstraction translation and the time to execute code to test. The Alloy-alpha analyzer study represents the largest implementation that we tested and for this study TestEra took 25 seconds to check the 12 cases.

When TestEra detects a violation of the property being tested, TestEra generates concrete counterexamples. In case no violation is detected, we can increase our confidence in the implementation by generating test inputs using a larger number of objects. Simply increasing the bound on input size and regenerating inputs produces some test inputs that have already been used in the smaller bound. TestEra's performance in such a case can be improved by ruling out inputs that can be found in a smaller bound; notice however that the number of inputs can grow exponentially with size so this improvement may only be marginal. As an illustration, the tabulated results are for linked lists with exactly three nodes (using up to 3 integers) and red-black trees with exactly five nodes (using up to 5 integers).

Notice that there is no need to generate all test cases first and then perform testing. If disk space is an issue, TestEra can perform testing and checking as the inputs are generated without having to store them. Selecting this trade-off, however means that these inputs cannot be reused to test other implementations or methods.

The INS and Alloy-alpha Analyzer studies were performed using the Alloy-alpha analyzer. For these studies we wrote by hand the Java code to perform abstractions and concretizations; for other studies the presented analysis is fully automatic. For manual translations, it took us a few hours (less than 8 man-hours) to write them; these translations were straightforward simply because we could define an isomorphism between structures represented by the Alloy model and those represented by the Java implementation.

To illustrate the nature of translations that we wrote by hand let us consider the INS study. Recall the 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.

It is worth noting that translations used by TestEra are at the input datatype level and therefore independent of the body of the method that is being tested. In other words, the same translations can be used to test several different methods that operate on a given set of input datatypes.

8. Discussion

We next discuss 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. Primitive types

To allow users to build specifications that include primitive types, we need to provide Alloy models for key operations on those types. 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 result of adding two integers. Consequently, the correctness specification must also mention a scope that is sufficiently large; determining such a scope (e.g., for sequences of integer operations) can be non-trivial. However, 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. Our current implementation provides limited support for `integer` and `boolean` types, including library code that automatically generates formulas for common integer operations, given an input scope.

8.2. Exceptions, arrays, multi-threadedness etc.

Treatment of exceptions and arrays is straightforward. Our current implementation does not support these. Also, we do not currently support automatic initialization of field values (e.g., for `final` fields), which requires building an Alloy model corresponding to the Java code.

Dynamically checking 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 [34]).

In object-oriented programs, inheritance plays a fundamental role. So far we have not addressed how to utilize class hierarchy in test generation. We present a systematic treatment of inheritance in a first-order setting in [41]; we have not yet implemented this approach.

8.3. Ease of specification

Even though use of path expressions is intuitive and allows building succinct formulas, use of (nested) quantifiers in building a specification can provide a learning challenge for most developers who are not adept at programming in a declarative style. We have addressed this limitation by providing a variety of (static and dynamic) analyses for the same specifications to make it more attractive for users to adopt our notation (see [32] for details.)

A key issue with building specifications that define complex structures is to correctly write constraints that precisely define the desired structure. For example, a tree structure may be intuitively easy to see, to state the constraints in a formal notation, however, is non-trivial. There are various strategies we can use to enhance our confidence in the specification: 1) the Alloy toolset 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 [50]; for these structures, users can simply compare the number of structures enumerated by the analyzer with the published numbers. We used these strategies for building specifications for the presented studies; we built each specification within a day's work (less than 8 man-hours).

We have recently explored adding common patterns to the specification language [40], such as acyclicity along a set of fields. These patterns not only allow users to easier specify common data structure properties, but also allow faster generation of structures [40]. 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 an input constraint.

8.4. Black-box testing

It is worth emphasizing that TestEra performs black-box [6] (or specification-based) testing. In other words, when testing a method the implementation of the method is not used in test generation. Since TestEra generates inputs from an input constraint, the constraint can be manually "strengthened" to generate "interesting" counterexamples, e.g, to rule-out generation of inputs that witness previously identified bugs—this also allows to identify new bugs without having to fix the ones previously discovered.

8.5. Systematic testing on an initial input segment

TestEra's approach to systematic testing is to exhaustively 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 some inputs. A strategy TestEra users can apply is to test exhaustively on all small inputs and test selectively on a few larger inputs to gain more confidence about the correctness of the program. Clearly, this strategy can be guided by the standard test adequacy criteria, such as structural code coverage. We plan to investigate such a strategy in the future. Users can also gain confidence by iteratively increasing the bound and testing on larger inputs as permitted by time constraints.

To determine how program coverage varies with bound on input size, we have recently [40] tested several benchmarks with the Korat framework [7] (see Section 9 for more details on Korat). The experiments show that it is feasible to achieve full statement and branch coverage for several data-structure benchmarks by testing on all inputs within a small input size.

8.6. Generating structurally complex data

A key aspect of TestEra's test input generation is that the inputs represent structurally complex data. Such data cannot be feasibly generated at the representation level merely by a random (or even brute force) assignment of field values as the number of valid structures with respect to the number of candidate structures tends to zero. Also, generating such data at the abstract level by a sequence of construction sequence can be inefficient; for example, to generate all red-black trees with 10

nodes we may require 10! (or about 3.6×10^6) sequences[10] whereas there are only 240 nonisomorphic red-black trees with 10 nodes. Efficient generation of structurally complex data requires systematic constraint solving techniques, such as those provided by TestEra and Korat.

9. Related work

We first discuss how TestEra relates to other projects on specification-based testing. We then compare TestEra with static analysis (and in particular model checking of software); although TestEra performs testing, i.e., dynamic analysis, it does so exhaustively within a given scope, which makes it related to some static analyses.

9.1. Specification-based testing

There is a large body of research on specification-based testing. An early paper by Goodenough and Gerhart [20] emphasizes its importance. Many projects automate test case generation from specifications, such as Z specifications [51, 24, 52, 17], UML statecharts [45, 44], ADL specifications [47, 8], or AsmL specifications [22, 21]. These specifications typically do not consider structurally complex inputs, such as linked data structures illustrated in TestEra's case studies. Further, these tools do not generate Java test cases.

Recently, the AsmL Test Generator (AsmLT) [18] was extended to handle structurally complex inputs using a Korat-like technique (see below). The first version of AsmLT [21] was based on finite-state machines (FSMs): an AsmL [22] specification is transformed into an FSM, and different traversals of FSM are used to construct test inputs. Dick and Faivre [16] were among the first to use an FSM-based approach: their tool first transforms a VDM [29] 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 [2] and BZTT [37]. These tools readily handle sequence of method calls, whereas we used TestEra only for testing one method at a time; more precisely, the method under test can contain a sequence of method calls, but we did not use TestEra to generate such sequences. However, with the exception of AsmLT, the other tools were not applied for structurally complex inputs such as those in TestEra's case studies.

Cheon and Leavens developed jmlunit [9] for testing Java programs. They use the Java Modeling Language (JML) [36] for specifications;

[10] Each sequence represents one particular order of insertion of 10 elements into an empty red-black tree.

jmlunit automatically translates JML specifications into test oracles for JUnit [5]. 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.

We have recently developed Korat [7], a framework that uses JML specifications and automates both test case generation and correctness checking. Korat uses a novel search algorithm to systematically explore the input space of a given Java predicate and generate all nonisomorphic inputs (within a given input size) that satisfy the predicate. Korat monitors each execution of the predicate on candidate inputs and prunes the search based on the fields accessed during the execution. We have primarily used Korat for black-box testing [7], but it can be also used for white-box testing [40]. In black-box testing, the predicate given to Korat represents the method precondition, and thus inputs that satisfy the predicate represent valid inputs for the method under test.

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. Regarding the ease of specification, we have a small experience with 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 sensitive to the actual way a specification is written, whereas generation in TestEra is relatively insensitive: for any two equivalent specifications, TestEra takes about the same time to generate inputs. Our experiments on several common data structures [7] showed that Korat, with JML specifications written to suit Korat, generates inputs faster than TestEra. Further, Korat is amenable to the use of dedicated generators [40] that make the generation even faster, while making the specifications easier to write. However, a specification not written to suit Korat could make generation in Korat much slower than in TestEra.

9.2. Static analysis

Several projects aim at developing static analyses for verifying program properties. The Extended Static Checker (ESC) [15] uses a theorem

prover to verify partial correctness 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, tools like ESC cannot verify complex properties of linked data structures (such as invariants of red-black trees). There are some recent research projects that attempt to address this issue. The Three-Valued-Logic Analyzer (TVLA) [46, 39] is the first static analysis system to verify that the list structure is preserved in programs that perform list reversals via destructive updating of the input list. TVLA has been used to analyze programs that manipulate doubly linked lists and circular lists, as well as some sorting programs. The pointer assertion logic engine (PALE) [43] can verify a large class of data structures that can be represented by a spanning tree backbone, with possibly additional pointers that do not add extra information. These data structures include doubly linked lists, trees with parent pointers, and threaded trees. While TVLA and PALE are primarily intraprocedural, Role Analysis [35] supports compositional interprocedural analysis and verifies similar properties.

In summary, 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, but for inputs bounded by a given size.

Vaziri and Jackson propose Jalloy approach [28, 53] for analyzing methods that manipulate linked data structures. Their approach is to first build an Alloy model of bounded segments of computation sequences and then check the model exhaustively with the Alloy Analyzer. Jalloy provides static analysis, but it is unsound with respect to both the size of input and the length of computation. TestEra not only checks the entire computation, but also handles larger inputs and more complex data structures than Jalloy. Further, unlike Jalloy, TestEra does not require specifications for all (helper) methods.

9.3. Software model checking

There has been a lot of recent interest in applying model checking to software. JavaPathFinder [54] and VeriSoft [19] operate directly on a Java, respectively C, program and systematically explore its state to check correctness. Other projects, such as Bandera [10] and JCAT [14], translate Java programs into the input language of existing model checkers like SPIN [23] and SMV [42]. They handle a significant portion

of Java, including dynamic allocation, object references, exceptions, inheritance, and threads. They also provide automated support for reducing program's state space through program slicing and data abstraction. SLAM [3, 4] 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 projects developed tools that offer strong guarantees in this area: if a program is successfully checked, there is no input/execution that would lead to an error. However, these projects 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.

10. Conclusion

TestEra is a novel framework for automated testing of Java programs. The key idea behind TestEra is to use structural invariants for input data to automatically generate test inputs. As an enabling technology, TestEra uses the first-order relational notation Alloy and the Alloy Analyzer. The automatic constraint solving ability of the Alloy Analyzer is used to generate concrete inputs to a program. The program is executed and each input-output pair is automatically checked against a correctness criteria expressed in Alloy. TestEra requires no user input besides a method specification and an integer bound on input size. A precise statement of a desired input-output relationship is something that any formal (automated) correctness checking framework requires.

We presented experimental results from several programs that were efficiently analyzed by TestEra. In all the cases, the analysis completed in less than a minute for the given small input bounds. When a program violates a correctness property, TestEra generates concrete counterexamples.

The experiments show that TestEra provides efficient enumeration of structurally complex data, which can be hard to systematically generate otherwise. Systematic testing on an initial segment of input space presents an exciting technique for finding errors. We believe TestEra's approach of modeling data but not computation promises scalability and wide application. We plan to extend TestEra's analysis to also report on structural code coverage, which would help users decide

when the program has been sufficiently tested. We also plan to evaluate TestEra on other programs.

Acknowledgements

We would like to thank the anonymous referees for their comments that helped us significantly improve our exposition. We would also like to thank Glenn Bruns and Patrice Godefroid for their comments and support. Daniel Jackson, Viktor Kuncak, Martin Rinard, and Ilya Shlyakhter provided valuable insights and discussions. Omar Abdala, Basel Al-Naffouri, and Faisal Anwar helped us with implementation and were among the first TestEra users. This work was funded in part by ITR grant #0086154 from the National Science Foundation. The work of the first author was done partly while visiting Bell Laboratories.

References

1. William Adjie-Winoto, Elliot Schwartz, Hari Balakrishnan, and Jeremy Lilley. The design and implementation of an intentional naming system. In *Proc. 17th ACM Symposium on Operating Systems Principles (SOSP)*, Kiawah Island, December 1999.
2. 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.
3. 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.
4. 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.
5. Kent Beck and Erich Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7), July 1998.
6. Boris Beizer. *Software Testing Techniques*. International Thomson Computer Press, 1990.
7. 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.
8. 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.
9. 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.

10. 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.
11. Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.
12. 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.
13. Markus Dahm. Byte code engineering library. `http://bcel.sourceforge.net/`.
14. C. Demartini, R. Iosif, and R. Sisto. A deadlock detection tool for concurrent Java programs. *Software - Practice and Experience*, July 1999.
15. David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, 1998.
16. 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.
17. 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.
18. Foundations of Software Engineering, Microsoft Research. The AsmL test generator tool. `http://research.microsoft.com/fse/asml/doc/AsmLTester.html`.
19. 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.
20. J. Goodenough and S. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, June 1975.
21. 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.
22. Yuri Gurevich. Evolving algebras 1993: Lipari guide. In *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.
23. Gerald Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), May 1997.
24. Hans-Martin Horcher. Improving software tests using Z specifications. In *Proc. 9th International Conference of Z Users, The Z Formal Specification Notation*, 1995.
25. Daniel Jackson. Micromodels of software: Modelling and analysis with Alloy, 2001. `http://sdg.lcs.mit.edu/alloy/book.pdf`.
26. Daniel Jackson. Alloy: A lightweight object modeling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2), April 2002.
27. 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.

28. 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.
29. Cliff B. Jones. *Systematic Software Development using VDM*. Prentice-Hall, 1990.
30. 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.
31. 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.
32. 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.
33. 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.
34. 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.
35. 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.
36. 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).
37. Bruno Legeard, Fabien Peureux, and Mark Utting. Automated boundary testing from Z and B. In *Proc. Formal Methods Europe (FME)*, Copenhagen, Denmark, July 2002.
38. Tal Lev-Ami, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Putting static analysis to work for verification: A case study. In *Proc. International Symposium on Software Testing and Analysis*, 2000.
39. Tal Lev-Ami and Mooly Sagiv. TVLA: A system for implementing static analyses. In *Proc. Static Analysis Symposium*, Santa Barbara, CA, June 2000.
40. 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.
41. Darko Marinov and Sarfraz Khurshid. VAlloy: Virtual functions meet a relational language. In *Proc. Formal Methods Europe (FME)*, Copenhagen, Denmark, July 2002.
42. K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
43. 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.

44. Jeff Offutt and Aynur Abdurazik. Generating tests from UML specifications. In *Proc. Second International Conference on the Unified Modeling Language*, October 1999.
45. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley Object Technology Series, 1998.
46. 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.
47. 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.
48. Elliot Schwartz. Design and implementation of intentional names. Master's thesis, MIT Laboratory for Computer Science, Cambridge, MA, June 1999.
49. Ilya Shlyakhter. Generating effective symmetry-breaking predicates for search problems. In *Proc. Workshop on Theory and Applications of Satisfiability Testing*, June 2001.
50. 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`.
51. J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, second edition, 1992.
52. Phil Stocks and David Carrington. A framework for specification-based testing. *IEEE Transactions on Software Engineering*, 22(11):777–793, 1996.
53. 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.
54. 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.