TestEra: A Novel Framework for Automated Testing of Java Programs

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

Abstract

We present TestEra, a novel framework for automated testing of Java programs. TestEra automatically generates all non-isomorphic test cases, within a given input size, and evaluates correctness criteria. As an enabling technology, TestEra uses Alloy, a first-order relational language, and the Alloy Analyzer. Checking a program with TestEra involves modeling the correctness criteria for the program in Alloy and specifying abstraction and concretization translations between instances of Alloy models and Java data structures. TestEra produces concrete Java inputs as counterexamples to violated correctness criteria. This paper discusses TestEra's analyses of several case studies: methods that manipulate singly linked lists and red-black trees, a naming architecture, and a part of the Alloy Analyzer.

1. Introduction

Manual software testing, in general, and generation of test data, in particular, are labor-intensive processes. Automated testing can significantly reduce the cost of software development and maintenance [3].

TestEra is a novel framework that automates both generation of test data and evaluation of correctness criteria for Java programs. As an enabling technology, TestEra uses the first-order relational language Alloy [15] and the Alloy Analyzer (AA) [12,14]. AA provides automatic analysis of Alloy specifications by generating instances that satisfy the constraints expressed in the specification.

The key idea behind TestEra is to use Alloy to express the structural invariants of inputs and the correctness criteria for a Java program, but not the program's computation. In previous work [17] on analyzing a naming architecture, we modeled both inputs and computation in Alloy. We discovered that manually modeling computation is complicated due to Alloy's declarative nature and lack of support for recursion. Automatically modeling computation in Alloy was performed [16] for a subset of Java, but the approach does not seem to scale at present.

In TestEra, a specification of inputs to a Java program is built in Alloy. TestEra uses AA to automatically generate, for a given input size, all non-isomorphic [27] instances for that specification. Next, TestEra uses a *concretization* to translate these instances to Java inputs, which form the test cases for the given program. The output produced by the program is mapped back into Alloy using an *abstraction*. Finally, TestEra uses AA to check the input and the output against a correctness criteria given in Alloy.

TestEra aims at detecting as many errors as possible without generating spurious error reports. This is achieved at the expense of compromising completeness. Errors may be missed, but reported errors are concrete counterexamples to violated properties.

To analyze a program, TestEra requires the correctness criterion for the program, as well as concretization (abstraction) translations for input (output) data structures. A precise correctness criterion is something that any correctness checking method requires. We have found translations to be straightforward to write, even for complex data structures.

Since TestEra automatically generates 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.

In this paper, we present the core components and analysis architecture of TestEra. We also show various applications of our prototype implementation of TestEra. We illustrate TestEra's capabilities by showing its analyses of not only intricate programs manipulating complicated data structures, but also complex tools like AA itself. TestEra was able to identify subtle bugs in a part of the AA.

The rest of the paper is organized as follows. Section 2 gives an example that illustrates the key steps of TestEra's analysis. In Section 3, we present the TestEra framework.

Section 4 discusses various case studies that we have performed to evaluate TestEra. In Section 5, we discuss our prototype implementation and its performance. We discuss related work in Section 6 and conclusions in Section 7.

2. Example

In this section we present an example that illustrates the key steps involved in TestEra's analysis. We also introduce the parts of Alloy notation that we use in this paper.

An Alloy specification is a sequence of paragraphs that either introduce new types or record constraints on *fields* of existing types. Alloy assumes a universe of atoms partitioned into subsets, each of which is associated with a basic type. Details of the notation can be found in [15] and of the Alloy Analyzer in [12, 14, 27].

Consider the following Java declaration of acyclic, singly linked lists and a (recursive) method for performing merge sort.

```
class List {
    int elem;
    List next;

    static List mergeSort(List l) {...}
}
```

Acyclic lists, the input to the method `mergeSort`, can be modeled in Alloy with the following specification.

```
module list
import integer

sig List {
  elem: Integer,
  next: option List }

fun Acyclic(l: List) {
  all n: l.*next | sole n.~next // At Most One Parent
  no l.~next }                  // Head Has No Parent

static sig Input extends List {}

fact GenerateInputs {
  Acyclic(Input) }
```

The declaration `module` names the specification, and `import` includes existing specifications into the current one. The *signature* declaration `List` introduces this uninterpreted type, along with functions `elem : List` $\rightarrow$ `Integer` and `next : List` $\rightarrow$ `List`. `next` is a partial function, indicated by the declaration `option`.

The Alloy *function* `Acyclic`, when invoked, constrains its input `l` to be acyclic. The dot operator '`.`' represents relational image; '`~`' and '`*`' denote transpose and reflexive transitive closure respectively.

The quantifier `all` stands for universal quantification. For instance, the constraint `all n:l.*next | F` holds if and only if evaluation of the *formula* `F` holds for each atom in the transitive closure of `next` starting from `l`. Formulas within curly braces are implicitly conjoined. The quantifier `sole` stands for "at most one". There are also quantifiers `some` and `no` with the obvious meaning.

The *subsignature* `Input` declares a subset of `List` that has exactly one atom, as constrained by the declaration `static`. `Input` models the input to `mergeSort`; the *fact* `GenerateInputs` constrains the atom in `Input` to be always acyclic by invoking the function `Acyclic` with argument `Input`. The fact need not be invoked itself, since it is required to always hold.

Given an Alloy specification, AA automatically finds *instances* that satisfy the specification, i.e., the valuations of relations and signatures that make all the facts in the specification true. AA finds instances within a pre-specified *scope*—the maximum number of atoms in each basic signature. AA can also enumerate all non-isomorphic instances.

Given the Alloy specification above, TestEra in the first phase generates all non-isomorphic test cases within a given input size: TestEra first uses AA to generate all non-isomorphic instances and then uses a *concretization translation* to translate Alloy instances to appropriate Java data structures.

In the next phase, TestEra executes `mergeSort` on each test case and abstracts the produced output using an *abstraction translation*. This abstracted output and the original input instance valuate the relations and signatures in the input/output Alloy specification. Finally, TestEra uses AA to check that this is a valid valuation.

The abstraction and concretization translations that TestEra uses are provided in Java. For the linked list data structure above, the translations are straightforward; we discuss them in Section 3.2.

The following Alloy code specifies the correctness criterion for `mergeSort` and is used in TestEra's phase 2 analysis. `Acyclic` and `Input` are as defined above. The Alloy operator '`#`' denotes cardinality (of sets).

```
fun Sorted(l:List) {
  // every element (except last) is not greater than next
  all n: l.*next | some n.next =>
                   n.elem <= n.next.elem }

fun Perm(l1:List, l2:List) {
  // same # of occurences in l1 and l2 for each integer
  all e: Integer | #(e.~elem & l1.*next) =
                   #(e.~elem & l2.*next) }

fun MergeSortOK(i:List, o:List) {
  Acyclic(o)
  Sorted(o)
  Perm(i,o) }

static sig Output ext List {}

fact OutputOK {
  MergeSortOK(Input, Output) }
```

The subsignature `Output` models the result produced by an invocation of `mergeSort`. The correctness criterion is expressed by the fact `OutputOK`, which essentially constrains the output to be acyclic, sorted, and a permutation of the input.

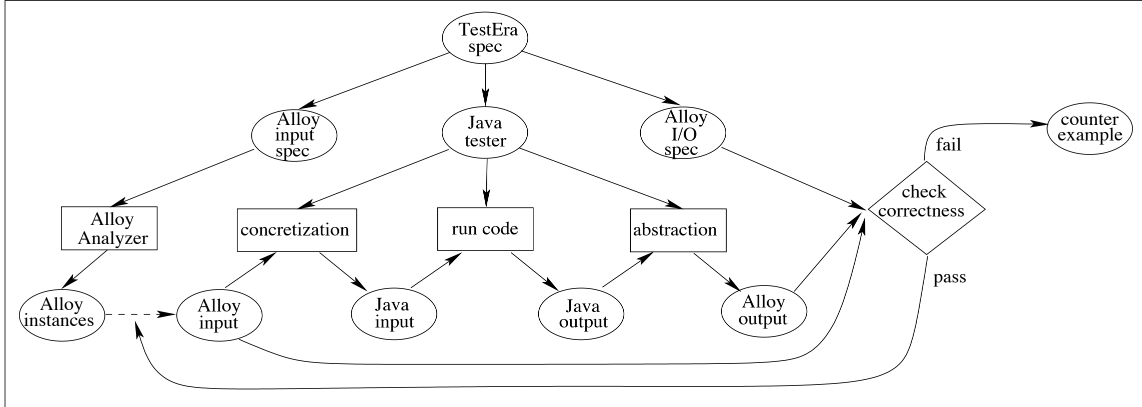

Figure 1. Basic TestEra framework

With phase 2 specification above, TestEra checks `mergeSort` and reports that there are no counterexamples for the correct implementation. However, if we erroneously reverse a comparison in the method for merging from `(l1.elem <= l2.elem)` to `(l1.elem >= l2.elem)`, TestEra detects violation of the correctness criterion and generates counterexamples, one of which is the following:

```
Counterexample found:
  Input List: 1 -> 1 -> 3 -> 2
  Output List: 3 -> 2 -> 1 -> 1
```

Most shape analysis techniques [19,20,22] are either unable to handle operations like `mergeSort` or require complicated invariants to be specified by the user for loops and method calls. We discuss this further in Section 6.

3. TestEra framework

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

3.1. Framework Outline

There are five key parts in testing a Java program with TestEra:

- Identifying a sequence σ of method calls to analyze.
- Creating an Alloy specification A_I of inputs to σ.
- Creating an Alloy specification $A_{I/O}$ of the correctness criteria for σ by relating an input to σ with the output of σ.
- Defining a concretization translation `a2j` from an Alloy instance of A_I to a Java input for σ.
- Defining an abstraction translation `j2a` from a Java output to a (partial) Alloy instance of $A_{I/O}$.

Figure 1 illustrates the main components of TestEra. A TestEra specification consists of a combination of Alloy and Java code. Three source files are extracted from this specification. Two of these files consist of Alloy specifications: one specification is for the inputs and the other specification is for the correctness criteria. The third file consists of Java code that translates input instances from Alloy to Java, runs the sequence of Java methods to test, and translates Java output back to Alloy.

TestEra's analysis has two phases:

- In the first phase, TestEra uses AA 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 translated, using concretization, into a Java input, which forms the test case for the sequence of method calls. Next, the output produced by the execution of this test case is translated, using abstraction, back into Alloy. This output Alloy instance and the original Alloy input instance valuate the signatures and relations of the Alloy input/output specification. AA then quickly evaluates if this valuation is valid. If the evaluation fails, TestEra reports a counterexample. If the evaluation passes, the next Alloy input instance is used for further testing.

3.2. Abstraction and Concretization Translations

We next discuss abstraction and concretization translations in TestEra. A concretization, abbreviated `a2j`, translates Alloy instances to Java data structures. An abstraction, abbreviated `j2a`, translates Java data structures to Alloy instances. In the current version of TestEra, all translations

between Alloy instances and Java data structures are actually written in Java.

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.

Recall the singly linked list example introduced in Section 2. We next give a2j translation for List, using pseudocode to simplify the presentation. In the pseudocode, we use foreach instead of Java for loops that iterate over collections, and we omit downcasting. We also use the pattern matching shorthand ⟨a,b⟩ to denote the Java object that represents a pair p, whose fields can be accessed using p.first() and p.second().

```
InputsAndMapAJ a2j(Instance i) {
  MapAJ map = new MapAJ();
  foreach (atom in i.getSigAtoms("List"))
    map.put(atom, new List());
  foreach (atom in i.getSigAtoms("Integer"))
    map.put(atom, new Integer(atom));
  List input = map.get("Input");
  foreach (<l,t> in i.getRelationMappings("elem"))
    map.get(l).elem = map.get(t).intValue();
  foreach (<l,t> in i.getRelationMappings("next"))
    map.get(l).next = map.get(t);
  return new InputsAndMapAJ(new Object[]{input}, map);}
```

For the input Alloy instance i, first a Java object corresponding to each atom in the signature List is created. Next, for each atom in signature Integer, a new java.lang.Integer object is created with the appropriate value. List variable input is then set to the appropriate object. This completes the first stage of a2j.

In the second stage, for each tuple in the Alloy relation elem, the corresponding field is appropriately set for the Java objects, and likewise tuples in relation next are processed. This completes the a2j translation.

Translations use the class MapAJ to store bi-directional mapping between Alloy atoms and Java objects. This class behaves like java.util.HashMap. For an object mapAJ of MapAJ, mapAJ.get(atom) returns the Java object corresponding to atom; mapAJ.getAtom(object,sig) returns the Alloy atom corresponding to (Java) object if there is such an atom in the mapping; if there is no atom, getAtom creates a new atom for the signature sig, adds it to the mapping, and returns it.

```
// execution of a test case
List output = mergeSort(input);
```

For each test case, TestEra invokes mergeSort. This simple execution is given above in Java. Below, we give the abstraction.

```
Instance j2a(Object output, Instance ret, MapAJ map) {
  List l = (Object) output;
  if (l==null) {
    ret.setSig("Output", null);
    return ret; }
  ret.setSig("Output", map.getAtom(l, "List"));
  Set visited = new HashSet();
  while ((!visited.contains(l)) && (l!=null)) {
    ret.addRelationMapping("elem",
          map.getAtom(l, "List"),
          map.getAtom(new Integer(l.elem), "Integer"));
    if (l.next!=null)
      ret.addRelationMapping("next",
            map.getAtom(l, "List"),
            map.getAtom(l.next, "List"));
    visited.add(l);
    l = l.next; }
  return ret; }
```

The parameter ret represents an input instance that is modified by j2a to include the output so that AA can check the correctness criteria. setSig updates the given signature by inserting the given atom; addRelationMapping behaves in a similar fashion for updating relations. If the output list is null, Output is set to be the empty set. Otherwise, the while loop is used to traverse the list and valuate the relations elem and next.

The set visited keeps track of the nodes that have been traversed in the loop. This is necessary to prevent the translation to enter an infinite loop if mergeSort erroneously produces an output list that is not acyclic.

The abstraction and concretization translations are simple for most data structures. It is possible to easily automate the generation of translations whenever there is a direct correspondence between the description in Alloy and the Java declaration. However, when the correspondence is not obvious, TestEra requires user-specified translations.

4. 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.

4.1. 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 [2] 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 simple 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 in `java.util.TreeMap`. In particular, we illustrate TestEra's analysis of the `deleteEntry` method in class `java.util.TreeMap$Entry`, which is responsible for deleting a given node from the tree. Deletion is the most complex operation among the standard operations on red-black trees and involves rotations. Together with the auxiliary methods, deletion in `java.util.TreeMap` is about 300 lines of Java code.

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

```
public class TreeMap {
    Entry root;
    ...
    static final boolean RED   = false;
    static final boolean BLACK = true;
    static class Entry implements Map.Entry {
        Object key;
        Object value;
        Entry left;
        Entry right;
        Entry parent;
        boolean color;
        ...
    }
    public void deleteEntry(Entry p) {...}
    ...
}
```

TestEra automatically generates input red-black trees as test cases for `deleteEntry` using the Alloy specification of the red-black properties presented below. Since red-black trees in `java.util.TreeMap` implement a mapping between keys and values, an `Entry` has two data fields: `key` and `value`. The field `value` represents the value that the corresponding `key` is mapped to. We abstract the field `value`, because it is irrelevant for the `deleteEntry` method. In the actual specification, we also omit `key` and identify an entry with its data, i.e., treat the name of the entry atom, assigned by AA, as its key. For clarity of presentation, we leave `key` in here.

```
module redBlackTree
import integer

sig Color {}
partition static sig Red, Black ext Color {}

sig Entry {
  key: option Integer,
  color: Color,
  left: option Entry,
  right: option Entry,
  parent: option Entry }
sig Root ext Entry {}

static sig NIL ext Entry {} // models leaf nodes
```

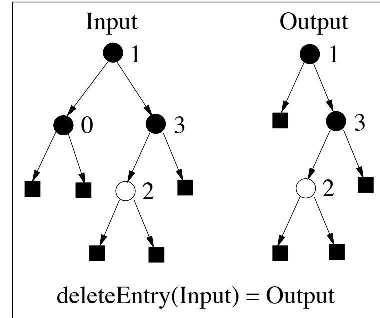

Figure 2. A counterexample for `deleteEntry`

```
fact ParentDefn {
  all e1, e2: Entry |
    e1 in e2.parent <=> e2 in e1.left + e1.right }

fun HasNILChild(e: Entry) { NIL in e.left + e.right }

fact RedBlackFacts {
  // every node is red or black -- holds by construction
  // leafs are black
  NIL.color = Black
  // red has black children
  all e: Entry | e.color = Red =>
                   (e.left + e.right).color = Black
  // paths from root to NIL have same # of black nodes
  all e1, e2: Entry |
    HasNILChild(e1) && HasNILChild(e2) =>
    #(e1.*parent & Black.~color) =
      #(e2.*parent & Black.~color) }

fact BinaryTreeFacts {
  // root has no parent
  no Root.parent
  // acyclic
  all e: Entry | e !in e.^(~parent)
  // unique children
  all e: Entry - NIL |
    e.left != e.right ||
    e.left + e.right = NIL
  // internal nodes
  all e: Entry - NIL |
    some e.left && some e.right && some e.key
  // leaf nodes
  no NIL.left && no NIL.right && no NIL.key }

fact SearchTreeFacts {
  // left subtree has smaller keys
  all e: Entry - NIL |
    all el: e.left.*(~parent) - NIL | el.key <= e.key
  // right subtree has larger keys
  all e: Entry - NIL |
    all er: e.right.*(~parent) - NIL | e.key <= er.key }
```

The four red-black properties are simply described in the fact `RedBlackFacts`. Since red-black trees are binary search trees, we also describe basic structural invariants that hold for binary trees in `BinaryTreeFacts` and the property that for any node, the elements in its left sub-tree are smaller than itself and those in the right sub-tree are larger than itself in `SearchTreeFacts`. ('^' denotes transitive closure, and '`||`' is logical or.)

To complete the phase 1 specification, we include the following declarations that model the input parameters of the `deleteEntry` method:

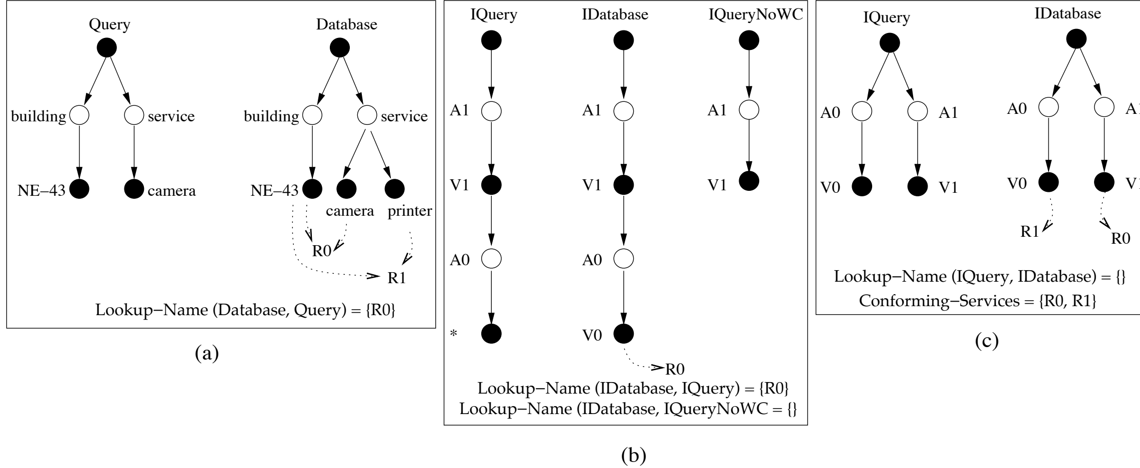

Figure 3. (a) Intentional names in INS, (b) and (c) counterexamples for `Lookup-Name`

```
static sig InputTreeRoot ext Root {}

static sig RemoveEntry ext Entry {}
```

After generating the test cases using the above specification, TestEra in phase 2 of its analysis tests `deleteEntry`. The Alloy specification for phase 2 contains constraints similar to the ones above and the correctness constraint that the set of keys in the output is equal to those in the input except for the key to delete.

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 2 presents a counterexample. 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 output tree is not balanced; more precisely, property 4 is violated.

The close correspondence between the Java data structures and the Alloy declarations in this case gives straightforward translations for abstraction and concretization. We do not present them here.

It should be noted here that Alloy provides an expressive notation for expressing properties of data structures. This is evident from the fact that the fourth property of red-black trees is not expressible in the logics used in [4, 22]. Similarly, [20] cannot check the `deleteEntry` method above.

4.2. 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 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 3(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 3(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 [26]. 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 [18]. The original Java implementation of INS [26] 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 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 3(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 3(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`. 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.

We define a service s as 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 with INS. The corrected algorithm now forms a part of the INS code base.

4.3. Alloy 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 Analyzer.

The bugs appear in AA 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 AA picks for an atom of a basic signature. These bugs have simple fixes and will be corrected in the upcoming release of the new version of AA.

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 AA 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 AA 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 disjoint a1,a2:Atom |
    a1.namePrefix != a2.namePrefix ||
    a1.nameSuffix != a2.nameSuffix }

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

partition static sig Signature, Test ext SigName {}
partition static sig S, T ext Prefix {}
partition static sig Zero, One ext 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 AA 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)}
```

Case Study	Property/Method Tested	Max size of a sig	Phase 1		Phase 2	
			#Tests	t[sec]	#Passed	t[sec]
Singly Linked Lists	`mergeSort`	4	42	10	42 (100%)	7
	`mergeSort (erroneous)`	4	42	0	0 (0%)	7
Red Black Trees	`deleteEntry`	5	54	31	54 (100%)	13
`java.util.TreeMap`	`deleteEntry (erroneous)`	5	54	0	30 (56%)	13
INS	`published wildcard claim`	3	12	9	10 (83%)	6
	`monotonicity of addition`	4	160	14	150 (93%)	9
	`Lookup-Name (original)`	3	16	8	10 (62%)	6
	`Lookup-Name (corrected)`	3	16	0	16 (100%)	6
AA	`disjoint sigs, unique atoms`	2	12	5	6 (50%)	25

Table 1. Summary of TestEra's analyses

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 AA, 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 AA:

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

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

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 AA. As mentioned, though, the violations can be easily fixed.

5. Implementation and Performance

We have implemented a prototype version[1] of TestEra in Java. Table 1 summarizes the performance of our implementation on the presented case studies; analyses were conducted on a Pentium III, 700 MHz processor. (All times are in seconds; `0sec` in phase 1 indicates reusing already generated tests.) In all the cases, TestEra takes less than a minute to complete both the automatic generation of instances and the verification of correctness.

[1] We have presented the specifications in the new version of Alloy [15]. The analyzer for this version has not been released yet. We performed the actual analyses on specifications written in an earlier version [13] and available at `http://www.mit.edu/~sarfraz/testera/`.

In phase 1, TestEra typically generates several test cases per second. Among the studies we have presented, the most complex structural invariants are those for red-black trees. This is indicated by the time TestEra takes to generate test cases. It should be noted, though, that the number of possible states to consider for generating red-black trees with five nodes is over 2^{80}. Of course, AA prunes away [14] most of these states, and that is why the test cases are generated fairly quickly.

In phase 2, TestEra's performance depends on the complexity of the code being executed. The is indicated by the last row in TestEra's analysis of AA, when it takes 25 seconds to test the 12 test cases. Each of these test cases requires using AA to generate instances and then checking those instances.

Writing abstraction and concretization translations in Java is straightforward in most cases. In all the cases we have presented, the translations were written within a few hours. We are also considering to introduce a special purpose language to ease development of these translations. For cases when there is a direct correspondence between an Alloy model and Java class declaration, we would like to automatically infer these translations. However, when the correspondence is more involved, such as for INS and AA, the translations may need to be specified by hand.

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 cases using a larger number of atoms in basic signatures. Simply increasing the scope and regenerating instances produces some test cases that have already been used in the smaller scope. TestEra's performance in such a case can be enhanced by ruling out instances that can be found in a smaller scope. As an illustration, we have ruled out some instances in the presented case studies. For example, we only considered linked lists with at least three integers and red-black trees with at least two nodes.

6. Related Work

There is a large body of research on specification-based testing. An early paper by Goodenough and Gerhart [8] argues its importance.

Horcher [11] presents a technique for software testing based on Z [28] specifications. This technique provides automated test execution and result evaluation. However, concrete input test data need to be selected manually from an automatically generated set of test classes.

The UMLTest tool [23] automatically generates tests from UML [24] statecharts and enabled transitions, but requires all variables to be boolean, among other limiting assumptions it makes about the UML input file. Applied to a C implementation of a cruise control, it detects several faults that were inserted by hand.

Chang et al. [5] present a technique for deriving test conditions—a set of boolean conditions on values of parameters—from ADL [25] specifications. These test conditions are used to guide test selection and to measure comprehensiveness of existing test suites.

Alloy has previously been used to analyze Java programs [16,17]. These techniques involve modeling both the input data structures and computation in Alloy. An Alloy model of the core computation of the Intentional Naming System is manually built in [17]. This manual translation of non-trivial imperative code into a declarative language is extremely subtle, and required a week of careful thinking. In this approach, modifications to an implementation require manual remodeling of computation.

The approach for analyzing bounded initial segments of computation sequences manipulating linked lists in [16] presents a way to automatically build an Alloy model of computation. This technique, however does not appear to scale. This is because Alloy does not have built in support for recursion, control flow, or sequencing of operations, which makes it hard to model imperative code.

The Pointer Assertion Logic Engine (PALE) [22] can verify partial specifications of data structures that can be expressed as *graph types*, and produce counterexamples for violated properties. However, this technique requires complicated, explicitly stated invariants, since its automatic verification works only for loop free code. Graph types cannot express arbitrary graphs. For example, it is not possible to say that every node in a graph has a pointer to some other node—a property easily expressible in Alloy. Also, the notation used in PALE lacks expressivity to state various interesting properties of data structures. For example, it is not possible to directly state the fourth invariant of red-black trees in that notation, as noted by the inventors of PALE.

The Three-Valued-Logic Analyzer TVLA [20] performs fixed point iterations on abstract descriptions of the store, and can verify various shape properties. TVLA, however, cannot produce concrete counterexamples when programs fail to verify. TVLA has been used to verify sorting programs [19]. But it neither handles recursive sorting programs, nor can verify (due to data abstractions) if the *data* values in the output list are the same as those in the input list. Most shape analysis techniques, including PALE, lack the ability to check correctness of properties regarding data values. However, the partial correctness properties they verify hold for all inputs to the program.

We reported our initial experience with TestEra at the recent Software Model Checking Workshop [18]. We presented there only the INS case study. This paper describes TestEra in more detail and presents several case studies.

There has been a lot of recent interest in applying model checking to software. The Bandera project [6] provides automated support for reducing a program's state space through program slicing and data abstraction. It allows users to analyze properties of Java programs by modeling them into the input language of one of several existing model checkers like SPIN [10] and SMV [21]. JavaPathFinder [9] and JCAT [7] transliterate Java programs into Promela programs. They handle a significant portion of Java including dynamic object allocation, object references, exception processing, and inheritance.

Most of the work on applying model checking to software has focused on analyzing event sequences and not data structures. Where data structures have been analyzed, the purpose has been to simplify the state machine to be analyzed and not to provide the user feedback about the data structures themselves.

7. Conclusions and Future Work

TestEra is a novel framework for automated testing of Java programs. The key idea behind TestEra is to use structural invariants on input data to automatically generate test cases and then test the output produced by a program against a correctness criteria.

As an enabling technology, TestEra uses the first-order relational notation Alloy and the Alloy Analyzer. The automatic constraint solving ability of AA is used in conjunction with a concretization translation to generate inputs to a program. The program is executed and each output produced is abstracted using an abstraction translation. Finally, each input-output pair is automatically verified against a correctness criteria expressed in Alloy. A precise statement of a desired input-output relationship is something that any verification method requires.

We presented several programs that were efficiently analyzed by TestEra. In all the cases, the analysis completed in a couple of seconds. When a property is refuted, TestEra generates concrete counterexamples.

We believe that the approach taken by TestEra promises

scalability and wide application since computation is not modeled and can be arbitrarily complex. The translations required for input and output are usually straightforward. The same translation is used for testing *any* program manipulating inputs of a particular datatype. We are considering exploring the correctness properties of the translations and even to automate them.

To help the user decide that the program is tested enough, we are planning to extend TestEra's analysis to report functional coverage. We would also like to evaluate TestEra on other programs.

Acknowledgements

We would like to thank Glenn Bruns and Patrice Godefroid for their comments and support. We would also like to thank Daniel Jackson, Viktor Kunčak, Martin Rinard, and Ilya Shlyakhter for helpful discussions. This work was funded in part by ITR grant #0086154 from the National Science Foundation. The work of the second author was done partly while visiting Bell Laboratories.

References

[1] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *Proc. 17th ACM Symposium on Operating Systems*, Kiawah Island, Dec. 1999.

[2] R. Bayer. Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, 1, 1972.

[3] B. Beizer. *Software Testing Techniques*. International Thomson Computer Press, 1990.

[4] M. Benedikt, T. Reps, and M. Sagiv. A decidable logic for describing linked data structures. In *Proc. European Symposium on Programming*, Amsterdam, The Netherlands, Mar. 1999.

[5] J. Chang, D. Richardson, and S. Sankar. Structural specification-based testing with ADL. In *Proc. Third International Symposium on Software Testing and Analysis*, January 1996.

[6] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proc. 22nd International Conference on Software Engineering*, June 2000.

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

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

[9] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, 1999.

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

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

[12] D. Jackson. Automating first-order relational logic. In *Proc. ACM SIGSOFT Eighth International Symposium on Foundations of Software Engineering*, San Diego, CA, Nov. 2000.

[13] D. Jackson. Alloy: A lightweight object modeling notation. *ACM Transactions on Software Engineering and Methodology*, 2001. To appear.

[14] D. Jackson, I. Schechter, and I. Shlyakhter. ALCOA: The Alloy constraint analyzer. In *Proc. International Conference on Software Engineering*, Limerick, Ireland, June 2000.

[15] D. Jackson, I. Shlyakhter, and M. Sridharan. A micromodularity mechanism. In *Proc. 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Vienna, Austria, September 2001. To appear.

[16] D. Jackson and M. Vaziri. Finding bugs with a constraint solver. In *Proc. International Symposium on Software Testing and Analysis (ISSTA 2000)*, Portland, OR, Aug. 2000.

[17] S. Khurshid and D. Jackson. Exploring the design of an intentional naming scheme with an automatic constraint analyzer. In *Proc. 15th IEEE International Conference on Automated Software Engineering*, Grenoble, France, Sept. 2000.

[18] S. Khurshid and D. Marinov. Checking Java implementation of a naming architecture using TestEra. In *Post-CAV Workshop on Software Model Checking*, volume 55(3) of *Electronic Notes in Theoretical Computer Science (ENTCS)*, Paris, France, July 2001. Elsevier Science.

[19] T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification: A case study. In *Proc. International Symposium on Software Testing and Analysis*, 2000.

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

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

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

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

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

[25] 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.

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

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

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