Electronic Notes in Theoretical Computer Science 55 No. 3 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume55.html 21 pages

Checking Java Implementation of a Naming Architecture Using TestEra

Sarfraz Khurshid and Darko Marinov

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

Abstract

TestEra is a novel framework for automated testing of Java programs. We have built TestEra upon Alloy, a lightweight first-order relational language, and the Alloy Analyzer, a fully automatic simulation and checking tool. Checking a Java 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 automatically generates all non-isomorphic test cases within a given input size and verifies the correctness.

We present our initial evaluation of TestEra performed by checking the Java implementation of a naming architecture for resource discovery in dynamic networked environments. Our study delineates the use of TestEra in testing methods for manipulating complex data structures.

1 Introduction

Software testing is a very labor intensive and expensive problem. Studies indicate that software testing consumes more than fifty percent of the cost of software development [3]. As software becomes more pervasive and is used more often to perform critical tasks, it will be required to have higher quality. The percentage of development costs devoted to testing will increase significantly unless more efficient ways to perform effective testing are found.

Generation of test data that satisfy testing requirements is a particularly time consuming component of testing. Automating test data generation can significantly reduce the cost of software development and maintenance.

TestEra [17] is a novel framework for automated testing of Java programs. TestEra is built upon Alloy [15], a lightweight first-order relational language suitable for expressing structural properties of software, and the Alloy Analyzer (AA) [12, 14], a fully automatic simulation and checking tool. The key

idea behind TestEra is to use Alloy to model the structure of inputs and outputs and the correctness criteria for a Java program, without modeling the program's computation.

TestEra can be used to test functional correctness of sequences of method calls. Checking with TestEra provides automatic generation of test data and evaluation of a given correctness criteria for each output produced. In TestEra, an abstraction model of inputs to a Java program is built in Alloy. AA is used to automatically generate all non-isomorphic [22] instances for that model within a given input size. A *concretization* is used to translate these instances to Java inputs, which form the test cases for the given program. An *abstraction* is used to translate back into Alloy the program's output, which is then verified by AA against the correctness criteria expressed in Alloy.

In this paper, we explore the use of TestEra by applying it to the Java implementation of the Intentional Naming System (INS) [1, 21], a recently proposed naming architecture for resource discovery and service location in dynamic networked environments. We perform several analyses on the INS implementation and illustrate the use of TestEra in testing properties of complex data structures and methods for manipulating them.

Our previous analysis [16] of INS using the Alloy Analyzer involved modeling both the correctness criteria and the computation in Alloy. Modeling computation was the most subtle aspect, because it involved manual translation of non-trivial imperative code into a declarative language. It required a lot of careful thinking and took about a week to accomplish. Also, any changes to the implementation necessitated manual remodeling of the computation.

TestEra presents a solution to the problems encountered in modeling computation—TestEra executes the actual implementation code, exploiting Alloy's suitability for expressing structural properties to automatically generate test cases. TestEra provides a novel lightweight approach to formal methods. The abstraction and concretization translations for evaluating INS using TestEra were constructed in a day by a researcher (the second author), who had a limited experience with Alloy and no prior knowledge of INS code.

Our study has several significant aspects. First, it shows how to use a simple declarative language to perform automated testing of complex Java programs. Second, our study illustrates the role a lightweight formal method can play in interactively correcting a flawed implementation without the need for modeling computation. Third, our study illustrates how easy it is for system builders to overlook subtle bugs that can be detected using fairly small inputs. It also shows how the lack of a formal specification can cause developers to develop incorrect intuition about the behavior of their system.

The rest of this paper is organized as follows. Section 2 presents an overview of the TestEra framework and the key notions of Alloy and AA. Section 3 describes the fundamentals of the naming architecture of INS. In Section 4 we apply the TestEra framework to test the Java implementation of INS. Related work is presented in Section 5, and we conclude in Section 6.

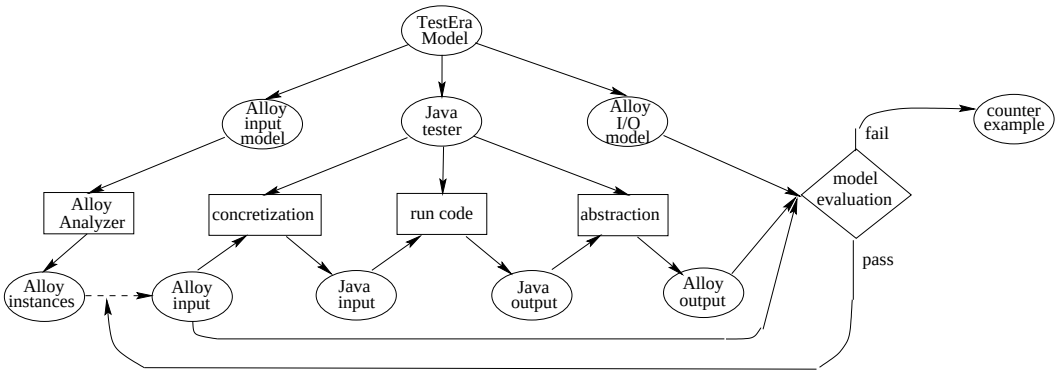

Fig. 1. TestEra Framework

2 TestEra

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

Alloy can be used in various approaches to analyze Java programs. One approach is to model both the inputs and the computation in Alloy and to use AA to verify the correctness criteria, also expressed in Alloy. Our earlier analysis [16] of INS proceeded in this fashion. We discovered that manually modeling computation is complicated due to Alloy's declarative nature. Further, this translation is hard to automate due to scalability issues of AA's automatic analysis.

TestEra takes a different approach to analyze Java programs—model correctness in Alloy, but test the actual Java code by verifying appropriately mapped inputs and outputs. TestEra exploits Alloy's suitability for expressing structural properties and uses AA to automatically generate test cases for the Java code.

2.1 The TestEra Framework

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

- Identifying a sequence σ of method calls to analyze.
- Creating an Alloy model A_I of inputs to σ.
- Creating an Alloy model $A_{I/O}$ that expresses the correctness criteria for σ, 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 model consists of a combination of Alloy and Java code. Three source files are extracted

from this model. Two of these files consist of Alloy models; one modeling the inputs and the other modeling 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, the Alloy input model is used by AA to generate all non-isomorphic Alloy instances.
- In the second phase, each of these 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 form the Alloy input/output model. AA then quickly evaluates if this model satisfies the correctness criteria. If the evaluation fails, a counterexample is reported. If the evaluation passes, the next Alloy input instance is used for further testing.

2.2 The Alloy Language

An Alloy model is a sequence of paragraphs. There are two kinds of paragraphs: signatures, used to construct new types, and a variety of formula paragraphs, used to record constraints. Before we describe these paragraphs we give a brief description of the syntax and semantics of formulas.

2.2.1 Relational Expressions

Alloy is a strongly typed language that assumes a universe of atoms partitioned into subsets, each of which is associated with a basic type. The standard mathematical notion of a relation is the only datatype. The value of any expression is always a relation, i.e., a collection of tuples of atoms. Each element of such a tuple is atomic and belongs to some basic type (i.e., the element cannot itself be a relation). A relation may have any arity greater than one. Relations are typed.

Sets are expressed as unary relations. Scalars and tuples are expressed as singleton relations.

Relations are 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.

There are two product operators: dot and arrow. The dot operator is composition: `p.q` is the relation containing the tuples $\langle p_0, \ldots, p_{m-1}, q_1, \ldots, q_n \rangle$, where `p` contains $\langle p_0, \ldots, p_m \rangle$, `q` contains $\langle q_o, \ldots, q_n \rangle$, and $p_m = q_0$. When `p` is a unary relation (i.e., a set) and `q` is a binary relation, the composition `p.q` is standard composition.

The arrow operator is cross product: `p->q` is the relation containing the tuples $\langle p_0, \ldots, p_m, q_0, \ldots, q_n \rangle$ when `p` contains $\langle p_0, \ldots, p_m \rangle$ and `q` contains

$\langle q_0, \ldots, q_n \rangle$.

There are three unary operators: `~` (transpose), `+` (transitive closure), and `*` (reflexive transitive closure). They can be applied only to binary relations, and they have their standard interpretation.

2.2.2 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`.

The standard logical operators are included: `&&` (conjunction), `||` (disjunction), `=>` (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`. A side-condition on the variable `v` might be added by embellishing the expression `e` with *multiplicity markings*.

There are two kinds of markings. A set marking is one of the keywords `set` or `scalar` prefixing the expression; `scalar` adds the side condition that the variable denotes a relation containing a single tuple, and `set` says it may contain any number of tuples. The default marking is `set`, except when the comparison operator is the colon (`:`) or negated colon (`!:`), and the expression on the right is unary, in which case it is `scalar`.

A relation marking is one of the symbols `!`, `?`, or `+` read *exactly one*, *at most one*, and *one or more* respectively. These markings are applied to the left and right of an arrow operator. Suppose a relation `r` is declared as

```
r : e1 m -> n e2
```

where `m` and `n` are relation markings. The markings are interpreted as imposing a side condition on `r` saying that for each tuple t_1 in `e1`, there are `n` tuples t_2 in `e2` such that $t_1 t_2$ appears in `r`, and for each tuple t_2 in `e2`, there are `m` tuples t_1 such that $t_1 t_2$ appears in `r`.

The declaration

```
partition 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 form a partition, i.e., share no tuple and their union is `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).

2.2.3 Signatures, Functions, and Facts

The signature declaration introduces a basic type and a collection of fields in it along with the types of the fields and constraints on their values. For example,

```
sig Value {}
```

introduces `Value` as an uninterpreted type with no fields. A signature may inherit fields and constraints from another signature. The construct

```
static sig Wildcard extends Value {}
```

declares `Wildcard` to be a subset of `Value` and inherit from it. The keyword `static` constrains `Wildcard` to contain exactly one element.

A function (`fun`) is a formula that can be "invoked" (imported) elsewhere and thus must be named. Given values of the arguments, this formula returns true/false or a relational value.

A `fact` is a formula that takes no arguments and need not be invoked explicitly; it is always true. Facts need not be named.

2.3 Alloy Analyzer

The Alloy Analyzer [1] (AA) [12, 14] is an automatic tool for analyzing Alloy models. Given a formula and a *scope*—a bound on the number of atoms in the universe—AA determines whether there exists a model of the formula (i.e., 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, AA's analysis cannot be a decision procedure. Instead, AA limits its analysis to a finite scope that bounds the size of the carrier sets of the basic types. A model is *within a scope of* k if it assigns to each type no more than k elements. AA's analysis [12] is based on a translation to a boolean satisfaction problem and gains its power by exploiting state-of-the-art SAT solvers.

AA provides two kinds of analysis: *simulation* in which the consistency of a fact or function is demonstrated by generating a snapshot showing its invocation, and *checking*, in which a consequence of the specification is tested by attempting to generate a counterexample.

AA can enumerate all possible instances of an Alloy model. AA adapts the symmetry-breaking predicates of Crawford et al. [6] to reduce 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 [22]. TestEra exploits this functionality to automatically generate test cases from an Alloy model of inputs to a Java program.

[1] In this paper, we use the new version of Alloy [15], for which the analyzer has not been released yet. We performed the actual analyses on models we wrote in Alloy-Alpha [13]. The models are available at `http://www.mit.edu/~sarfraz/testera/`.

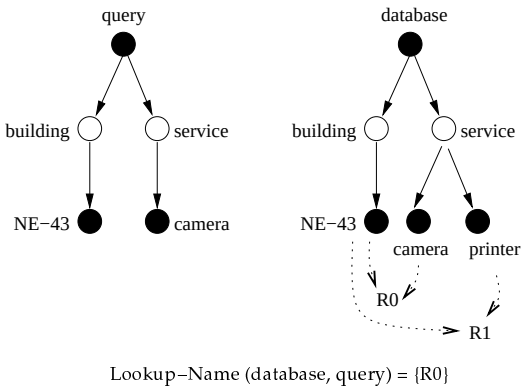

Fig. 2. Example of a query, database, and `Lookup-Name` operation

3 Intentional Naming System

One proposal for service discovery in dynamic networked environments is the *Intentional Naming System* (INS) [1, 21], which allows services to describe and refer to each other using *intentional* names. These names describe a set of properties that the services provide rather than the network location, e.g., "the nearest color printer that handles transparencies" rather than "printer643.lcs.mit.edu" or "18.31.0.99". This allows applications to specify in their queries *what* service they want, not *where* in the network topology it resides. Similarly, services advertise themselves using intentional names.

An intentional name is a tree of alternating levels of *attributes* and *values*. The query in Figure 2 is an example intentional name; hollow circles identify attributes and filled circles identify values. Attributes represent categories in which an object can be classified. Each attribute has a corresponding value that is the object's classification within that category. A wildcard may be used in place of a value to show that any value is acceptable. The query describes an object in building NE-43 that provides a camera service. The database stores two objects: R0 that provides a camera service in building NE-43 and R1 that provides a printer service in the same building.

Queries made by clients are resolved by calling `Lookup-Name` method. Invoking `Lookup-Name` on the example query and database returns R0, because the value of attribute "service" sought by the client (i.e., camera) does not match that provided by R1 (i.e., printer).

When a service advertises its availability to a name resolver, the service is included in the database stored by that name resolver using the `Add-Name` method. To periodically update adjacent name resolvers about new or expired services, a name resolver extracts advertisements from its database using the `Get-Name` method, and conveys them to its neighbors.

A Java implementation of the naming architecture of INS appears in [21]. About 1500 lines of Java code implement the core functionality of INS. Appendix A shows relevant parts of the Java data structures and Appendix B gives the Java implementation of `Lookup-Name` provided in [21].

```
sig Attribute {}
sig Value {}
sig Record {}

static sig Wildcard extends Value {}

sig AVTree {
  root: Value,
  vnodes: set Value,
  anodes: set Attribute,
  av: anodes !->+ (vnodes - root),
  va: (vnodes - Wildcard) !-> anodes}
fact {all t: AVTree | with t |
  vnodes = root.*(va.av)
  && some root.va}

sig Query extends AVTree {}
fact {all q: Query |
  all a: q.anodes | one a.(q.av)}
```

```
sig DB extends AVTree {
  records: set Record,
  recs: (vnodes - root) +-> records}
fact {all d: DB | with d |
  // wildcard not in database
  Wildcard !in vnodes
  // leaf values contain a record
  all v: vnodes - dom va | some v.recs
  // records at lowest possible level
  all v: vnodes |
    no v.recs & v.+(~av.~va).recs
  // in advertisements,
  // each attribute has one child
  all a: anodes |
    all disj v1, v2: a.av |
      some rr = *(va.av).recs |
        no v1.rr & v2.rr
        partition Query, DB: AVTree}
```

Fig. 3. Alloy model of INS query and database (Phase 1)

4 TestEra's Analysis of INS

We use TestEra to test various properties of INS methods. These properties can be classified into the following three categories:

- essential for correctness, e.g., addition to database does not reduce results so that the availability of new services alone does not reduce the options a client has to acquire his desired functionality;
- sufficient for correctness, e.g., name resolution returns exactly the conforming services;
- INS inventors' claims, e.g., correspondence of wildcards to missing attributes in name resolution.

To use TestEra to test INS, we need to identify the methods we want to test, construct Alloy models of inputs and correctness criteria, and define abstraction and concretization translations between instances of Alloy models and INS Java data structures.

We focus our analysis of INS on `Lookup-Name`, the most important method in the naming architecture. It takes as input a query and a database, so the first step is to model these input data structures in Alloy.

4.1 Modeling Intentional Names in Alloy

Figure 3 shows our Alloy model of intentional names. `Attribute`, `Value` and `Record` introduce these basic types. `Wildcard` is a special `Value`.

An intentional name is modeled as a rooted tree with alternating levels of attributes and values using two relations, `av` and `va`, that represent the

```
class Attribute {
    String attribute; ... }
class Value {
    String value; ... }
class AttributeNode {
    Attribute a;
    Vector children; // Vector of ValueNode
    ValueNode parent;
    void addValueNode(ValueNode vn); ... } // add a child
class ValueNode {
    Value v;
    Vector children; // Vector of AttributeNode
    AttributeNode parent;
    NameRecordSet routeSet; // 'recs' in the model
    void addAttributeNode(AttributeNode an); ... } // add a child
class NameRecord { // 'sig Record' in the model
    static int next_id = 0;
    final int id;
    Vector parents; ... } // Vector of ValueNode
class NameTree extends ValueNode { // 'sig DB' in the model
    // v from ValueNode is always null
    Vector nameRecords; ... } // all NameRecords in the tree
```

Fig. 4. Java data structures implementing a database

edges from attributes to values and values to attributes, respectively. For each `AVTree` `t`, `t.root` is a special `Value` designated as its root, `t.vnodes` is a subset of `Value` that is the set of all values in `t`, `t.anodes` is a subset of `Attribute` that is the set of all attributes in `t`, `t.av` is a relation mapping `t.anodes` to elements in `t.vnodes` other than `t.root`, and `t.va` is a relation mapping `t.vnodes` other than `Wildcard` to `t.anodes`. The multiplicity markings on `t.av` require that for each non-root element `v` in `t.vnodes`, there is exactly one element in `t.anodes` that is mapped to `v` by the relation `t.av`, and each element in `t.anodes` is mapped to at least one non-root element in `t.vnodes`. Similarly, the `!` marking on `t.va` enforces that for each `a` in `t.anodes`, there is exactly one `v` in `t.vnodes - Wildcard` that maps to `a`. The last constraint states that all values in `t.vnodes` are reachable from `t.root`.

A `Query` is an `AVTree` that has the additional constraint that each attribute is paired up with exactly one value.

A `DB` is an `AVTree` that introduces two new fields. For each `DB` `db`, `db.records` is the set of all records in `db`, and `db.recs` is a relation that maps non-root values in `db.vnodes` to records stored there. A group of constraints expresses a valid database structure.

4.2 Abstraction and Concretization Translations

We next present abstraction and concretization translations for INS. A concretization, abbreviated `a2j`, translates Alloy instances to Java data structures. An abstraction, abbreviated `j2a`, translates Java data structures to Al-

```
class INSTester extends Tester {
    InputsAndMapAj a2j(Instance i) { // concretization
        MapAJ map = new MapAJ();
        // create Java objects for the Alloy atoms from all basic types
        foreach (atom in i.getSigAtoms("Attribute"))
            map.put(atom, new AttributeNode(new Attribute(atom.name())));
        NameTree database = new NameTree();
        Atom dbRoot = i.getAtom("IDB.root");
        foreach (atom in i.getSigAtoms("Value")) {
            if (atom.equals(dbRoot)) map.put(atom, database);
            ... // similar exception for the root of the queries
            else map.put(atom, new ValueNode(new Value(atom.name())));
        }
        foreach (atom in i.getSigAtoms("Record"))
            map.put(atom, new NameRecord());
        // create Java database corresponding to the Alloy database
        foreach (<vAtom, aAtom> in i.getRelationMappings("IDB.va"))
            map.get(vAtom).addAttributeNode(map.get(aAtom));
        foreach (<aAtom, vAtom> in i.getRelationMappings("IDB.av"))
            map.get(aAtom).addValueNode(map.get(vAtom));
        foreach (<vAtom, rAtom> in i.getRelationMappings("IDB.recs")) {
            NameRecord nr = map.get(rAtom);
            map.get(vAtom).addNameRecordHere(nr);
            database.addNameRecord(nr);
        }
        ... // create Java queries corresponding to the Alloy queries
        return new InputsAndMapAJ(new Object[]{database, query}, map);
    }
    ... // j2a and testing methods
}
```

Fig. 5. Translation of a database from Alloy to Java

loy instances. For INS, the translations are between instances of the presented Alloy model of intentional names and the Java data structures implementing intentional names.

To illustrate translations, we describe the concretization of an INS database. Figure 4 shows the relevant part of Java classes implementing a database. The class `NameTree` represents a database. Each object of this class is the root of a name tree for the database, and it also contains a set of all records in the database. The nodes of the name tree are objects of classes `ValueNode` and `AttributeNode`, each of which contains a `Value` or an `Atribute`. Additionally, each value node that corresponds to a leaf of some advertisements stores a set of records for those advertisements.

Figure 5 shows the relevant part for the concretization of the input database, `IDB`. In the current version of TestEra, all translations between Alloy instances and Java data structures are written in Java. In the pseudo-code, we use `foreach` instead of Java `for` loops that iterate over collections, and we omit downcasting. We also use a pattern matching abbreviation `<a,b>` to denote a Java object `p` representing a pair whose elements can be accessed using `p.first()` and `p.second()`.

The example concretization 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 `map`). The only non-trivial part in the example is that the value corresponding to the root of the database is mapped to a `NameTree`, whereas all the other values are mapped to a `ValueNode`. In the second stage, `a2j` establishes the relationships between the Java objects created in the first stage and builds the data structures. In our running example, `a2j` only links the value and attribute nodes using the existing methods from their corresponding classes.

Translations use the class `MapAJ` to store bi-directional mapping between Alloy atoms and Java objects. This class behaves like `java.util.HashMap`; e.g., `map.get(atom)` returns the Java object corresponding to `atom`. (Abstractions use method `map.getAtom(object,sig)` that 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.)

4.3 Testing Properties

INS inventors did not formally specify the correctness criteria for database operations. However, they stated partial correctness properties, including claims about the functionality of the system. In this section we show a range of analyses that TestEra can perform.

Recall that a TestEra model provides three source files: an Alloy model of inputs, an Alloy model of correctness, and Java code for abstraction and concretization translations and the sequence of method calls to test. The analyses below use identical abstraction and concretization translations and the base Alloy models for inputs, but the correctness models in Alloy depend on the sequence of method calls to test. We also demonstrate how TestEra can be used to generate "interesting" inputs.

4.3.1 Wildcard Claim

The published description of `Lookup-Name` claims [1]:

> This algorithm uses the assumption that omitted attributes correspond to wildcards; this is true for both the queries and advertisements.

More formally, for all databases `db`, and all queries `q`, it should be that `Lookup-Name(db,q)` = `Lookup-Name(db,q')`, where `q'` is the same as `q` except that any wildcarded attributes in `q` do not appear in `q'`. The first property we test using TestEra is to evaluate this claim.

Figure 6, together with Figure 3, provides our Alloy model for generating inputs to test the claim about wildcards. We model three inputs. `IDB` models the input database, `IQuery` models a query, and `IQueryNoWC` is the same as `IQuery` with its wildcarded attributes removed.

```
static sig IDB extends DB {}

static sig IQuery extends Query {}
fact {Wildcard in IQuery.vnodes}

static sig IQueryNoWC extends Query {}
fact {all q: IQueryNoWC | with q |
  vnodes = IQuery.vnodes - Wildcard      // wildcard value removed
  anodes = IQuery.anodes -               // wildcarded attribute removed
    Wildcard.~(IQuery.av)
  av = IQuery.av - Attribute -> Wildcard // edge to wildcard removed
  va = IQuery.va -                       // edge to wildcarded attribute
    Value -> Wildcard.~(IQuery.av)}      // removed
```

Fig. 6. Generating inputs for wildcard claim (Phase 1)

In phase 1 of TestEra's analysis, AA is used to produce all such non-isomorphic inputs. Each input produces a test case that invokes two calls to `Lookup-Name`, one on `IDB` and `IQuery`, and the other one on `IDB` and `IQueryNoWC`, with all inputs mapped in Java.

The following model expresses the correctness condition for the claim:

```
sig LookupQuery extends Record {}
sig LookupQueryNoWC extends Record {}
fact WildcardClaim {LookupQuery = LookupQueryNoWC}
```

where `LookupQuery` is assigned the value `Lookup-Name(IDB`^{J}`,IQuery`^{J}`)`^{A} and `LookupQueryNoWC` the value `Lookup-Name(IDB`^{J}`,IQueryNoWC`^{J}`)`^{A}; the superscripts J and A indicate applications of the concretization and abstraction translations, respectively. The correctness criteria `WildcardClaim` is simply to test the values for equality.

We use a scope of 3 `Attribute`s, 3 `Value`s, and 2 `Record`s in phase 1, and AA automatically generates 12 tests in 9 seconds. The testing in phase 2 completes in 6 seconds and the INS implementation passes 10 tests. One of the failed tests is illustrated in Figure 7 and provides a counterexample to the INS's inventors claim.

It is conceivable that while making the claim about wildcards, INS's inventors considered only the case when query has attributes that exist in the database and values that have matching values in the database. It is easy to include this precondition on inputs with the following `fact`:

```
fact IQueryMatchesIDB {
  // attributes in query are in database
  IQuery.anodes in IDB.anodes
  // non-wildcard values in query are in database
  IQuery.vnodes - Wildcard in IDB.vnodes
  // (a,v) edges to non-wildcard values in query are in database
  all a: IQuery.anodes - Wildcard.~(IQuery.anodes) |
      a.(IQuery.av) in a.(IDB.av)
  // (v,a) edges in query are in database
  all v: IQuery.vnodes | v.(IQuery.va) in v.(IDB.va)}
```

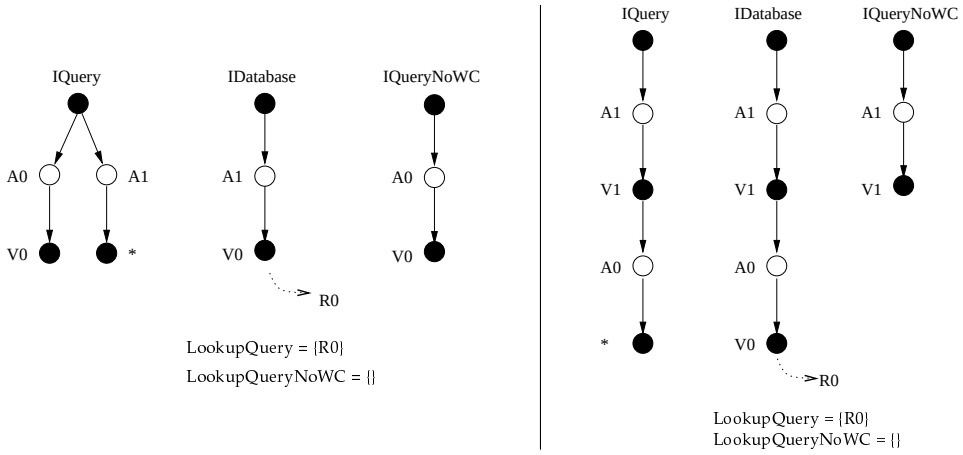

Fig. 7. TestEra counterexamples to `WildcardClaim`, without (left figure) and with (right figure) precondition that `IQuery` uses attributes and values that are in `IDB`

We add this `fact` to the input model shown in Figure 6 and use TestEra to test the wildcard claim in this special case. TestEra takes 10 seconds to produce 16 test cases using 2 `Attribute`s, 4 `Value`s, and 2 `Record`s in phase 1. The phase 2 analysis of this model takes 6 seconds, and TestEra refutes the wildcard claim with the special precondition, reporting that only 6 tests pass. Figure 7 shows one of the counterexamples. In general, removing wildcarded attributes from a query can both increase or decrease the number of services returned by `Lookup-Name`, whereas the inventors of INS assumed it would have no effect.

4.3.2 Monotonicity of Addition

Next we show a slightly different use of TestEra. We evaluate a partial correctness property of INS that involves the operations `Lookup-Name`, `Add-Name`, and `Get-Name`. For a query `q` and an advertisement `a`, INS does not specify how to evaluate whether a service `r` advertising `a` provides the properties desired by an application querying for `q`. A fair assumption is that `a` *conforms* to `q` if `Lookup-Name(db,q)` = `r`, where `db` is the database containing only the advertisement `a` made by `r`. We use this assumption as a basis to test whether `Lookup-Name` returns exactly those services whose advertisements conform to the query.

For input generation, we model only one database and one query:

```
static sig IDB extends DB {}
static sig IQuery extends Query {}
```

Together with Figure 3, this forms the phase 1 model for TestEra's analysis. For each record `r` in `IDB`, we extract the relevant advertised name `a` and create a new database `db` that consists of only the advertisement `a` and the record `r`. Then we check whether $\texttt{Lookup-Name(db,IQuery}^{J}\texttt{)}^{A} = \texttt{r}$. We collect all records `r` for which the previous equality holds to form a set `Conf` of services in `IDB` whose advertisements conform to `IQuery`. `Conf` is then compared for

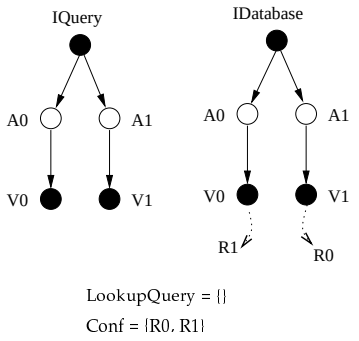

Fig. 8. TestEra counterexample to `PartialCorrectness`

equality with the result of `Lookup-Name(IDB`^{J}`,IQuery`^{J}`)`^{A}, using the Alloy `fact PartialCorrectness`:

```
sig LookupQuery extends Record {}
sig Conf extends Record {}
fact PartialCorrectness {LookupQuery = Conf}
```

Using a scope of 2 `Attribute`s, 4 `Value`s, and 2 `Record`s in phase 1 of TestEra, AA produces 160 test cases in 14 seconds. It takes 9 seconds to evaluate these test cases in phase 2, and TestEra reports that 10 of them produce counterexamples. Figure 8 shows one of the counterexamples.

This counterexample points out two defects in the design of `Lookup-Name`. First, there is no consistent notion of conformance. Second, simply adding new services to the system can reduce results produced by `Lookup-Name`, i.e., addition is not monotonic in INS—a property that is essential for the correctness of any such naming scheme.

4.3.3 Conformance

Both properties of `Lookup-Name` tested above involved, in phase 2 of TestEra, a simple Alloy `fact` that tested two sets for equality—something that can be simply verified in Java. We next test a property for which the phase 2 analysis of TestEra is more involved and illustrates the benefits of using Alloy for expressing correctness criteria.

Having established that INS does not originally have a consistent notion of conformance between a query and an advertisement, we define this notion for INS. We regard an advertisement to conform to a query if the query is a subtree of the advertisement, i.e., the advertisement mentions all the attributes and values that make up the query in the correct order. With this definition of conformance, we test if INS inventors' implementation of `Lookup-Name` returns exactly the conforming records.

The phase 1 model for testing conformance is the same as the phase 1 model for testing monotonicity of addition. Figure 9 shows the phase 2 model for conformance. Notice that this model has no explicit `fact`s other than the correctness condition, since the inputs already satisfy their constraints.

`LookupQuery` is assigned the value `Lookup-Name(IDB`^{J}`,IQuery`^{J}`)`^{A}. Using

```
sig Attribute {}
sig Value {}
sig Record {}

static sig Wildcard extends Value {}

sig AVTree {
  root: Value,
  vnodes: set Value,
  anodes: set Attribute,
  av: anodes !->+ (vnodes - Root),
  va: (vnodes - Wildcard) !-> anodes}

sig Query extends AVTree {}

sig DB extends AVTree {
  records: set Record,
  recs: (vnodes - root) +-> records}

static sig IDB extends DB {}
static sig IQuery extends Query {}
```

```
sig LookupQuery extends Record {}

fun Conforms(db: DB, q: Query,
             r: Record) {
  with db |
    some r' = r.~recs.*(~av.~va) {
    // all (v,a) edges in query
    // also in advertisement
    all a: q.anodes |
      a.~(q.va) = a.~va && a in r'.~av
  // the same for (a,v) edges
    all v: q.vnodes - Wildcard |
      v.~(q.av) = v.~av && v in r'}}

fact CorrectLookup {
  // soundness and completeness
  all db: DB | all q: Query |
    all r: db.records |
      Conforms(db,q,r) <=>
      r in LookupQuery}
```

Fig. 9. `Lookup-Name` input/output model (Phase 2)

a scope of 3 `Attribute`s, 3 `Value`s, and 2 `Record`s in phase 1 of TestEra, AA produces 16 test cases in 8 seconds. Phase 2 analysis takes 6 seconds and reports that only 10 pass. This is not surprising, since we are testing the original implementation of `Lookup-Name` for correctness against our definition of conformance. Nonetheless, doing so illustrates the diverse kinds of analyses allowed by TestEra.

We use the flaws in `Lookup-Name` identified thus far to correct its implementation with respect to the sub-tree definition of conformance given in `Conforms`. The test cases generated for evaluating `CorrectLookup` above are re-used to test this modified implementation. Phase 2 analysis of TestEra concludes in 6 seconds and results in no counterexamples.

4.4 Summary of results

Table 1 summarizes the tests performed on INS using TestEra: published wildcard claim, published wildcard claim with query in database, monotonicity of addition, correctness of the original `Lookup-Name`, and correctness of the modified `Lookup-Name`. All properties that are refuted do not require large inputs. The exhaustive (up to isomorphism) testing performed by TestEra uncovers subtle bugs that can be found using small inputs but went undetected for over a year of use of INS.

All the times shown in Table 1 include the time to boot up the Alloy Analyzer (about 4 sec). It is simple to perform TestEra's analysis by just starting up AA once to amortize this constant overhead. In this study we have not done so, because it does not add any insights into TestEra's analysis.

	Input size			*Phase 1*		*Phase 2*	
Property tested	*Val*	*Att*	*Rec*	*# Tests*	*Time*	*# Tests passed*	*Time*
Wildcard	3	3	2	12	9	10 (83%)	6
Wildcard-in	4	2	2	16	10	6 (37%)	6
Monotonocity	4	2	2	160	14	150 (93%)	9
Original	3	3	2	16	8	10 (62%)	6
Corrected	3	3	2	16	8	16 (100%)	6

Table 1
Summary of TestEra's analyses. (All times are in seconds.)

Recall that AA compiles Alloy models into boolean formulas and uses off-the-shelf SAT solvers to generate instances (counterexamples). Hence, it comes as no surprise that phase 2 analysis of TestEra takes lesser time than phase 1—in phase 2, a correctness condition is verified for a given valuation of relations in the Alloy model, whereas in phase 1, a valuation is generated; cf. verifying that a given solution makes a boolean formula true as opposed to finding a solution.

5 Related Work

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

Horcher [11] presents a technique for software testing based on Z [23] 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 [19] automatically generates tests from UML [20] 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 created by hand.

Chang et al. [4] present a technique for deriving test conditions—a set of boolean conditions on values of parameters—from Assertion Definition Language (ADL) specifications. These test conditions are used to guide test selection and to measure comprehensiveness of existing test suites.

Our previous analysis [16] of INS with Alloy involved modeling both the data structures and computation in Alloy. Modeling computation was the most subtle aspect of that model since it is hard to model recursion, control flow, and sequencing of operations in Alloy.

There has been a lot of recent interest in applying model checking to software. The Bandera project [5] 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 [18].

JavaPathFinder [9] and JCAT [7] transliterate Java programs to Promela programs. They handle a significant portion of Java including dynamic object allocation, object references, exception processing, and inheritance.

In the SLAM [2] project, sequential C programs are abstracted into boolean programs, and symbolic execution is used to map abstract counterexamples to concrete executions. The model checking algorithm of SLAM uses context-free grammar reachability to handle recursive procedure calls.

Most of the work on applying model checking to software has focused on analyzing event sequences, and when data structures were analyzed, the purpose was to simplify the resulting models, not to provide the user feedback about the data structures themselves. In contrast, TestEra focuses on testing complex properties of data structures.

6 Conclusions

TestEra provides a novel framework for automated testing of Java programs. Checking a Java program using TestEra requires modeling the input/output data structures and correctness criteria in Alloy, as well as writing abstraction and concretization translations, but does not require modeling the actual computation.

Writing translations for TestEra is much simpler than manually modeling computation in Alloy, because Alloy has no built-in support for advanced programming constructs. For INS, the translations were developed in a day by the second author, who had no prior knowledge of INS code and only a limited experience with Alloy. In comparison, INS computation had been modeled in Alloy in about a week by the first author, who at the time had a similar experience with INS and Alloy. Note also that the concretization translation is required even when the computation is manually modeled, in order to verify the counterexamples produced by AA with respect to the implementation.

Automatically modeling computation in Alloy is feasible, but the approach does not seem to scale at present. We instead plan to investigate approaches that automate writing of abstraction and concretization translations. As the first step, we are considering to introduce a special purpose language for writing the translations.

Non-isomorphic inputs to a Java program may be generated by writing an application-specific Java program. However, defining a model of inputs in Alloy and the concretization translation is much simpler.

TestEra's analysis of INS produced counterexamples to each property refuted in just a few seconds. This compares favorably with our previous analysis of INS where both the inputs and computation were modeled in Alloy. Additionally, TestEra produces counterexamples both in the form of Alloy

instances, which can be graphically viewed using AA's graphical component, and Java inputs. This makes it easier to debug the code.

In this paper, we presented the initial evaluation of TestEra by using it to check the Java implementation of the Intentional Naming System, a recently proposed naming architecture for resource discovery and service location in dynamic networked environments. We performed a wide range of analyses that illustrate the use of TestEra in testing properties of complex data structures and methods for manipulating them. Based on the encouraging results from this evaluation, we believe that TestEra can be efficiently used in testing other Java data structures. We plan to apply TestEra to more Java programs to determine how practical and scalable our approach is.

Acknowledgements

We would like to thank Daniel Jackson, Viktor Kunčak, Martin Rinard, Radu Rugină, Alexandru Sălcianu, Ilya Shlyakhter, and Manu Sridharan for helpful discussions and feedback. This work was funded in part by ITR grant #0086154 from the National Science Foundation.

References

[1] Adjie-Winoto, W., 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, 1999.

[2] Ball, T. and S. Rajamani, *Automatically validating temporal safety properties of interfaces*, in: *Proc. 8th International SPIN Workshop on Model Checking of Software*, 2001.

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

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

[5] Corbett, J., 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*, 2000.

[6] Crawford, J., 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.

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

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

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

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

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

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

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

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

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

[16] Khurshid, S. 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, 2000.

[17] Marinov, D. and S. Khurshid, *TestEra: A novel framework for automated testing of Java programs* (2001), submitted for publication.

[18] McMillan, K., "Symbolic Model Checking," Kluwer Academic Publishers, 1993.

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

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

[21] Schwartz, E., "Design and Implementation of Intentional Names," Master's thesis, MIT Laboratory for Computer Science, Cambridge, MA (1999).

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

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

A Concrete Data Types for INS [21]

```
class Attribute {
    String attribute;
    ... }
class Value {
    String value;
    boolean wildcard;
    // Representation Invariant:
    // wildcard == true -> String == null
    // wildcard != true -> String != null
    ... }
class AttributeNode {
    Attribute a;
    Vector children; // Vector of ValueNode
    ValueNode parent;
    void addValueNode(ValueNode vn); // add a child
    ... }
class ValueNode {
    Value v;
    Vector children; // Vector of AttributeNode
    AttributeNode parent;
    NameRecordSet routeSet; // 'recs' in the model
    void addAttributeNode(AttributeNode an); // add a child
    void addNameRecordHere(NameRecord r); // add a record to the set
    ... }
class NameRecord { // 'sig Record' in the model
    static int next_id = 0;
    final int id;
    Vector parents; // Vector of ValueNode
    ... }
class NameRecordSet {
    // if true, NameRecordSet contains all NameRecord objs
    boolean allRouteEntries;
    Vector routeEntries; // Vector of NameRecord
    // Representation Invariant:
    // if (allRouteEntries == true) routeEntries.isEmpty() = true;
    // routeEntries is sorted in ascending order of NameRecord.id
    ... }
class NameTree extends ValueNode { // 'sig DB' in the model
    // v from ValueNode is always null
    Vector nameRecords; // Vector of NameRecord
                        // (all NameRecords in the tree)
    void addNameRecord(NameRecord r); // add a record to the tree
    ... }
class AVelement {
    Attribute a;
    Value v;
    Vector children; // Vector of AVelement
    void addAVelement(AVelement c); // add a child
    ... }
class NameSpecifier extends AVelement { // 'sig Query' in the model
    // all variables inherited from AVelement, a and v not used
    ... }
```

B Lookup-Name implementation [21]

```
class NameTree extends ValueNode{
    NameRecord[] lookup(NameSpecifier s){
        NameRecordSet rs = super.lookup((AVPair)s);
        return (rs.toArray());
    } ...}
class ValueNode {
    NameRecordSet lookup(AVPair n){
        NameRecordSet S;
        // Start with S = the set of all possible route entries
        S = new NameRecordSet();
        S.addAllRouteEntries();
        // for each attribute-value pair p = (na,nv) in n
        for (Enumeration e1 = n.getAVPairs(); e1.hasMoreElements(); ) {
            AVPair p  = (AVPair)e1.nextElement();
            Attribute na = p.getAttribute();
            Value     nv = p.getValue();
            // Ta = the child of T (this) such that name(Ta) = name(na)
            AttributeNode Ta;
            try {
                Ta = findAttributeNode(na);
            } catch (ElementNotFound ex) {
                continue;
            }
            ValueNode Tv;
            if (nv.isWildcard()) {
                // Wildcard matching.
                NameRecordSet Sprime = new NameRecordSet();
                for (Enumeration e2 = Ta.getValueNodes();
                     e2.hasMoreElements();
                     ) {
                    Tv = (ValueNode)e2.nextElement();
                    Sprime.unionWith(Tv.routeSet);
                }
                S.intersectWith(Sprime);
            } else {
                // Normal matching.
                try {
                    Tv = Ta.findValueNode(nv);
                } catch (ElementNotFound ex) {
                    return(new NameRecordSet());
                }
                if (Tv.isLeaf() || p.isLeaf()) {
                    S.intersectWith(Tv.routeSet);
                } else {
                    S.intersectWith(Tv.lookup(p));
                }
            }
        }
        S.unionWith(routeSet);
        return(S);
    } ...}
```