TestEra: A Tool for Testing Java Programs Using Alloy Specifications

Shadi Abdul Khalek*, Guowei Yang*, Lingming Zhang*, Darko Marinov†, Sarfraz Khurshid*
*Electrical and Computer Engineering, University of Texas at Austin
Email: ak.shadi@utexas.edu, gyang@ece.utexas.edu, zhanglm@utexas.edu, khurshid@ece.utexas.edu
†Department of Computer Science, University of Illinois at Urbana-Champaign, Email: marinov@illinois.edu

***Abstract*—This tool paper presents an embodiment of TestEra – a framework developed in previous work for specification-based testing of Java programs. To test a Java method, TestEra uses the method's pre-condition specification to generate test inputs and the post-condition to check correctness of outputs. TestEra supports specifications written in Alloy – a first-order, declarative language based on relations – and uses the SAT-based back-end of the Alloy tool-set for systematic generation of test suites. Each test case is a JUnit test method, which performs three key steps: (1) initialization of pre-state, i.e., creation of inputs to the method under test; (2) invocation of the method; and (3) checking the correctness of post-state, i.e., checking the method output. The tool supports visualization of inputs and outputs as object graphs for graphical illustration of method behavior. TestEra is available for download to be used as a library or as an Eclipse plug-in.**

I. Introduction

The TestEra framework [1], [2] introduced *systematic*, black-box testing of Java programs using Alloy [3] specifications. Alloy is a declarative language based on sets and relations. The Alloy tool-set provides a fully automatic SAT-based analysis engine for solving Alloy formulas. TestEra uses Alloy as an enabling technology and provides *bounded-exhaustive testing*, where a program is tested against all (non-equivalent) inputs within a given input size. Such systematic testing was shown effective at finding bugs in various applications that take structurally complex test inputs [1], [4], including a resource-discovery system, a fault-tree analyzer, compilers, and refactoring engines.

The use of Alloy enables writing specifications at an intuitive abstract level. Solving pre-condition constraints provides generation of abstract inputs, which are concretized into actual inputs. Test execution runs the computation under test against these inputs to produce outputs. Correctness checking abstracts the outputs and checks them using the post-condition. Alloy's relational basis supports a natural view of the program heap: an edge-labeled graph with a set of nodes (one for each object) and a collection of edges (one for each field) [5].

TestEra was originally developed [1] for Alloy-alpha, an initial version of the Alloy language that had a flat representation of state, and later re-implemented [2] for Alloy 2.0, which introduced a hierarchical representation of state. Both these implementations were for in-house experiments and not released. This paper presents a tool embodiment of TestEra that supports Alloy 4.0 – the latest release of the Alloy tool-set – and is the first publicly available version of TestEra.

This paper makes the following contributions:

- **TestEra tool**. It presents the TestEra tool, which can be used as a library or installed as an Eclipse plug-in to generate JUnit test suites.
- **Annotations for test generation**. It presents Alloy annotations for Java programs to enable automated testing.
- **Visualization**. TestEra supports graphical illustration of executions of automatically generated tests using its data translation API and Alloy's visualization API.

II. Example

This section illustrates how programmers can use TestEra to test their programs. Consider a singly-linked list with a method `addNode` that appends a given value to the input list. Fig. 1 shows the annotated Java code that declares the list data structure and its method.

Our tool implementation introduces a "`@TestEra`" annotation for classes and methods (Section III). For a Java class, programmers can specify its class invariant that should be satisfied by all instances of the class. In our example, the invariant element of the annotation includes two universally quantified formulas. The first formula states directed acyclicity

```
@TestEra(invariant={
  "all l: LinkedList |
      all n: l.header.*next | n !in n.^next",
  "all l: LinkedList | l.size = #l.header.*next"} )
public class LinkedList {
  public ListNode header;
  public int size = 0;

  @TestEra(preCondition={"x >= 0",
    postCondition={"this.size' = this.size + 1"},
    runCommand="1 LinkedList, 3 ListNode, 3 int" )
  public void addNode(int x) {
    ListNode n = new ListNode();
    n.value = x;
    n.next = header;
    header = n;
    size++;
  }
}
public class ListNode {
  public int value;
  public ListNode next;
}
```

Fig. 1. TestEra annotated Java program of Linked List

```
@Test
public void test5() {
  // TestEra Auto-Comment: Initialization statements
  LinkedList LinkedList_0 = new LinkedList();
  LinkedList_0.size = 0;
  // TestEra Auto-Comment: Pre-state abstraction
  StateManager sm = new StateManager();
  sm.addToState("LinkedList_0", LinkedList_0);
  sm.generatePreState();
  // TestEra Auto-Comment: Invoke method under test
  LinkedList_0.addNode(0);
  // TestEra Auto-Comment: Post-state checking
  TestEra.checkPostState(
    sm, "dataStructures.list.LinkedList",
    "addNode", "LinkedList_0", "0");
}
```

Fig. 2. A JUnit test example

property by ensuring that a traversal that starts at a node cannot revisit that node. The keyword ‘all’ represents universal quantification and the dot operator ‘.’ represents relational join. An expression header.next represents the relational join of the next relation with the header element, which is equivalent to de-referencing the next field in the header object. The ‘*’ and ‘^’ operators represent reflexive transitive closure and transitive closure respectively. Thus, the expression header.*next represents the set of all nodes reachable from the header including the header element itself. The second invariant formula ensures that the size of the list is equal to the number of nodes reachable from the list’s header. We use the ‘#’ operator, which denotes set cardinality, to count the number of nodes reachable from the header. More details on ALloy are available elsewhere [3].

For a method under test, TestEra requires programmers to specify the method’s pre-condition and post-condition, and a bound on the input size, which is used in a run command to execute the Alloy Analyzer. For addNode method in the example, the pre-condition declares that the argument x is non-negative and the post-condition declares that the size of the list after executing the method is increased by 1. We use the ‘′’ operator to denote the post-state of the size field. The bound, defined in the runCommand annotation element, declares the scope of the LinkedList, ListNode, and primitive integers used in the tests to be generated.

Based on the Java code and the associated annotations, TestEra can automatically enumerate a number of JUnit tests within the user-specified scope. Fig. 2 shows an example generated JUnit test for the addNode method.

III. Annotation

This section describes the @TestEra annotation introduced by TestEra. Currently, we support the following five elements of the annotation:

1. isEnabled. It instructs TestEra whether to use the annotated field or class in the translation between Java and Alloy. The default value is true. For example, the annotation

```
@TestEra(isEnabled=false) int f;
```

specifies that the associated field f not be translated.

2. invariant. It applies only to classes and specifies the class invariants, which are translated into Alloy facts. For example, the annotation

```
@TestEra(invariant={
    "all l: LinkedList | l.size = #l.header.*next"})
```

specifies the invariant on the size and the nodes of the list. Multiple invariants are strings separated by commas.

3. preCondition. It applies only to methods and specifies the constraints that the test input must satisfy before executing the method. Similar to invariants, multiple constraints are strings separated by commas. For example, the annotation

```
@TestEra(preCondition={"x >= 0"})
```

specifies that the value of variable x must be non-negative.

4. postCondition. It applies only to methods and specifies the constraints that the method execution must satisfy in the post-state. The post-condition can specify a relationship between the post-state and pre-state of the method execution. For example, the annotation

```
@TestEra(postCondition={"this.size′=this.size + 1"})
```

specifies that size in the post-state is the size in the pre-state plus 1. Note that the ‘′’ character (apostrophe) denotes field traversal in the post-state. In contrast, fields without ‘′’ represent those in the pre-state. For non-static methods, the keyword this represents the receiver object of the method under test.

A post-condition can specify a constraint on the return value of the method invocation, where ‘\result’ is used to denote the return value. For example, the annotation

```
@TestEra(postCondition={"\result=this.header.value"})
```

specifies that the associated method should return the value of the header of the LinkedList.

5. runCommand. It applies only to methods and sets the scope for variables. For example, the annotation

```
@TestEra(runCommand="1 LinkedList, 3 ListNode, 3 int")
```

specifies the run command scope for the variables: 1 atom for LinkedList, 3 atoms for ListNode, and 3-bit integers (-4 to 3) for the int type.

IV. TestEra

A. Framework

In TestEra, programmers can describe class invariants and method pre- and post-conditions using the @TestEra annotations. Given a Java program and corresponding annotations, TestEra automates generation of JUnit tests – input generation and correctness checking. Fig. 3 shows the overview of TestEra steps. In the first step, TestEra takes the annotations and Java code, and translates them into corresponding Alloy models. Given a method under test, TestEra extracts its pre-condition and post-condition, and feeds the pre-condition together with the translated Alloy models to Alloy Analyzer, which uses

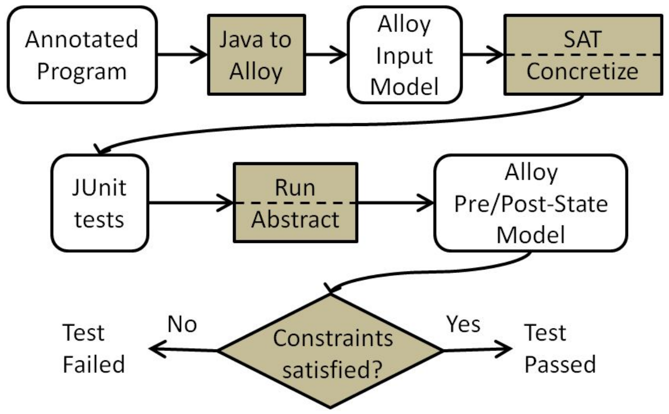

Fig. 3. Framework of TestEra

$$\begin{array}{lcl}
T(comUnit) & = & \textbf{module}\ unitID\ T(classDecl)^* \\
T(classDecl) & = & T(cAnnot)\ \textbf{sig}\ classID \\
 & & [\textbf{extends}\ classID] \\
 & & T(classBody) \\
T(classBody) & = & \{T(fieldDecl)^*\} \\
T(fieldDecl) & = & fieldID\ :\ T(jType) \rightarrow State \\
T(cAnnot) & = & \textbf{fact}\ \{invariants\} \\
T(jType) & = & classID \mid primitiveAlloy
\end{array}$$

Fig. 4. Translation rules for Java classes with TestEra annotations.

off-the-shelf SAT solvers, to generate Alloy instances which satisfy the pre-condition. Next, TestEra concretizes each generated Alloy instance into a JUnit test case. When the JUnit test case is executed, the runtime support of TestEra abstracts both the test inputs and the test outputs into Alloy pre/post-state model specifications. At the last step, TestEra checks if the model satisfied the post-condition constraints. Note that the model already contains the translated class invariant. A test passes when the post-condition is satisfied and fails otherwise.

B. Alloy Model Generation

The translation step of TestEra takes the Java code with TestEra annotations, and generates Alloy models as outputs. Fig. 4 shows the translation rules, where "[]" denotes optional elements, and "*x**" denotes a list of *x*s. According to the rules, TestEra translates each compilation unit into an analyzable *module* in Alloy. TestEra translates Java classes and their corresponding fields into Alloy signatures (`sig`) and their fields, respectively. TestEra translates user-defined invariant annotations into Alloy formulas that should be satisfied both for pre-state and post-states (`fact`). Finally, TestEra maps Java primitive types that have Alloy counterparts to those types, and maps non-primitive Java types to the corresponding Alloy signatures. Implementation-wise, TestEra creates a top-level directory for storing all the generated Alloy models, and for each Java compilation unit, TestEra creates an Alloy module with the same name and the same path under the top-level directory, which enables invocation of the Alloy tool-set as described in the next section.

$$\begin{array}{lcl}
T(methdDecl) & = & \textbf{module}\ methodID\_\mathrm{Test} \\
 & & T(mAnnot)^* \\
T(mAnnot) & = & T(preAnnot) | T(postAnnot) \\
 & & | T(runAnnot) \\
T(preAnnot) & = & \textbf{pred}\ methdID\_\mathrm{pre}\ ([this\ :\ classID,] \\
 & & (T(paraID)\ :\ T(jType)),^* \\
 & & [result\ :\ T(jType),])\{preConditions\} \\
T(postAnnot) & = & \textbf{pred}\ methdID\_\mathrm{post}\ ([this\ :\ classID,] \\
 & & (T(paraID)\ :\ T(jType)),^* \\
 & & [result\ :\ T(jType),])\{postConditions\} \\
T(runAnnot) & = & \textbf{run}\ methdID\_\mathrm{pre}\ \textbf{for}\ runCommand
\end{array}$$

Fig. 5. Translation rules for Java methods with TestEra annotations.

C. Alloy Input Generation

For each method under test, TestEra automatically creates an Alloy module that includes translated method annotations. For every pre-condition (or post-condition) annotation, TestEra creates a new predicate named $methodID\_\mathrm{pre}$ (or $methodID\_\mathrm{post}$, respectively) with appropriate constraints. In addition, TestEra adds the receiver object as the first parameter of the predicate if the method under test is not static, and adds a `result` parameter to represent the method's return value if it is not `void`. For a run-command annotation, TestEra adds a run command at the end of the generated module for running the $methodID\_\mathrm{pre}$ predicate. The run command is then executed by Alloy Analyzer to generate Alloy instances satisfying the pre-conditions of the module.

D. JUnit Test Case Generation

At this step, TestEra translates an Alloy instance into an executable Java unit test. This step involves several operations:
Concretization: TestEra translates an abstract Alloy instance into Java statements. First, for each object in the Alloy instance, TestEra requires that the object class has a default constructor and creates a statement that invokes the constructor. Then, for each relation in the Alloy instance, TestEra creates a Java field assignment. The current version requires that the corresponding field be `public`. TestEra also adds statements that store the pre-state of the objects. Next, TestEra adds the call to the method under test with the input parameters and (if needed) receiver object. Finally, TestEra adds a call to the TestEra post-state checker. Fig. 2 shows an example output.
Abstraction: This operation is performed while running the JUnit tests generated by TestEra. After executing the test, and storing the pre- and post-states of the objects, TestEra generates a new Alloy module by translating these object states into the Alloy representation. This translation generates relations for object fields for both pre-states and post-states.
Checking: For correctness checking, TestEra generates a post-state predicate that checks whether the post-conditions on the method output are satisfied. Alloy Analyzer is used to run the post predicate: if the formula is satisfiable, then the test passes; otherwise, the post-condition was not satisfied by the method, and thus the test fails. In case of a passing test, TestEra provides a visualization tool to view the change between pre- and post-states of the input objects.

V. Implementation

Our TestEra implementation has two basic components: the TestEra library that provides the core functionality, and the TestEra plug-in that provides a friendly user interface within Eclipse. TestEra is available online for download:

http://www.ece.utexas.edu/∼khurshid/svvat/projects/testera/

VI. Illustration

We illustrate the key steps of TestEra using the `LinkedList` example from Section II. Fig. 1 shows the annotations used to declare the invariants of the data structure and the constraints to generate test input for the `addNode` method. The first step TestEra performs is to translate Java classes into Alloy signatures; in this example, TestEra generates the following Alloy modules for `LinkedList` and `ListNode` classes:

```
module LinkedList
open ListNode
open util/State

sig LinkedList {
  header: ListNode lone-> State,
  size: Int one-> State
}
fact LinkedList_fact {
  all s:State {
 all l:LinkedList | all n : l.(header.s).*(next.s) |
      n !in n.^(next.s)
 all l:LinkedList | #l.(header.s).*(next.s) = l.(
     size.s)
  }
}

module ListNode
open util/State

sig ListNode {
  value: Int one-> State,
  next: ListNode lone-> State
}

module util/State
abstract sig State {}
one sig Pre extends State {}
```

The `State` signature represents the state of the relations in the model. Since class invariants hold true in all publicly visible states (i.e. pre- and post-states), we add a universal quantification "`all s:State`" to the invariant to ensure that.

For the example `addNode` method, TestEra translates the method annotations into the following Alloy predicate for input generation:

```
module LinkedList/addNode_PreTest
open LinkedList

pred LinkedList::addNode_pre(x: Int){
 x >= 0
}

run addNode_pre for 1 LinkedList, 3 ListNode, 3 int
```

Since, the `addNode` method is not static, we append the class type to the predicate signature to reflect the receiver object on which the method is called. Since the precondition holds true only in the pre-state, TestEra uses a *Pre* signature, that extends the abstract `State` signature, in the pre-condition constraints.

The Alloy Analyzer runs this Alloy module and generates instances satisfying the pre-conditions for the method under test. TestEra then concretizes these instances into concrete JUnit tests, an example of which is shown in Fig. 2.

While executing the JUnit test, TestEra stores the pre- and post-states of objects using the `StateManager`. TestEra then generates an Alloy module for checking whether the post-conditions are satisfied; the following shows the module for the JUnit test from Fig. 2:

```
module addNode_PostTest
open addNode_post
one sig ListNode_0 extends ListNode {}
one sig LinkedList_0 extends LinkedList {}

fact {
  no LinkedList_0.(header.Pre)
  LinkedList_0.(size.Pre) = 0
  LinkedList_0.(header.Post) = ListNode_0
  LinkedList_0.(size.Post) = 1
  ListNode_0.(value.Post) = 0
  no ListNode_0.(next.Post)
}

pred TESTERA(){
  LinkedList_0::addNode_post [0]
}
run TESTERA for 3 but exactly 1 ListNode,1
    LinkedList

module addNode_post
one sig Post extends State {}

pred LinkedList::addNode_post(x: Int){
  this.(size.Post) = this.(size.Pre) + 1
}
```

Finally, TestEra runs the post-condition model. If it is satisfiable, then the test passes; otherwise, the test fails. In this example, the test passes.

Acknowledgments

This material is based upon work partially supported by the National Science Foundation under Grant Nos. CNS-0958231, CNS-0958199, CCF-0845628, CCF-0746856, and IIS-0438967, and AFOSR grant FA9550-09-1-0351.

References

[1] D. Marinov and S. Khurshid, "Testera: A novel framework for automated testing of java programs," in *Proc. of International Conference on Automated Software Engineering (ASE)*, 2001.

[2] S. Khurshid, "Generating structurally complex tests from declarative constraints," Ph.D. dissertation, Massachusetts Institute of Technology, 2003.

[3] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[4] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov, "Test generation through programming in UDITA," in *Proc. of International Conference on Software Engineering (ICSE)*, 2010.

[5] V. Kuncak and D. Jackson, "Relational analysis of algebraic datatypes," in *Proc. of joint meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2005.