Augmenting Automatically Generated Unit-Test Suites with Regression Oracle Checking

Tao Xie

Department of Computer Science
North Carolina State University
Raleigh, NC 27695
xie@csc.ncsu.edu

Abstract. A test case consists of two parts: a test input to exercise the program under test and a test oracle to check the correctness of the test execution. A test oracle is often in the form of executable assertions such as in the JUnit testing framework. Manually generated test cases are valuable in exposing program faults in the current program version or regression faults in future program versions. However, manually generated test cases are often insufficient for assuring high software quality. We can then use an existing test-generation tool to generate new test inputs to augment the existing test suite. However, without specifications these automatically generated test inputs often do not have test oracles for exposing faults. In this paper, we have developed an automatic approach and its supporting tool, called Orstra, for augmenting an automatically generated unit-test suite with regression oracle checking. The augmented test suite has an improved capability of guarding against regression faults. In our new approach, Orstra first executes the test suite and collects the class under test's object states exercised by the test suite. On collected object states, Orstra creates assertions for asserting behavior of the object states. On executed observer methods (public methods with non-void returns), Orstra also creates assertions for asserting their return values. Then later when the class is changed, the augmented test suite is executed to check whether assertion violations are reported. We have evaluated Orstra on augmenting automatically generated tests for eleven subjects taken from a variety of sources. The experimental results show that an automatically generated test suite's fault-detection capability can be effectively improved after being augmented by Orstra.

1 Introduction

To expose faults in a program, developers create a test suite, which includes a set of test cases to exercise the program. A test case consists of two parts: a test input to exercise the program under test and a test oracle to check the correctness of the test execution. A test oracle is often in the form of runtime assertions [2, 36] such as in the JUnit testing framework [19]. In Extreme Programming [7] practice, writing unit tests has become an important part of software development. Unit tests help expose not only faults in the current program version but also regression faults introduced during program changes: these written unit tests allow developers to change their code in a continuous and controlled way. However, some special test inputs are often overlooked by developers and typical manually created unit test suites are often insufficient for assuring high software quality. Then developers can use one of the existing automatic test-generation tools [8, 11, 12, 31, 42–44] to generate a large number of test inputs to complement the manually created tests. However, without specifications, these automatically generated test inputs do not have test oracles, which can be used to check whether test executions are correct. In this paper, we have developed a new automatic approach that adds assertions into an automatically generated test suite so that the augmented test suite has an improved capability of guarding against regression faults.

Our approach focuses on object-oriented unit tests, such as the ones written in the JUnit testing framework [19]. An object-oriented unit test consists of sequences of method invocations. Our approach proposes a framework for asserting the behavior of a method invocation in an object-oriented unit-test suite. Behavior of an invocation depends on the state of the receiver object and method arguments at the beginning of the invocation. Behavior of an invocation can be asserted by checking at the end of the invocation the return value of the invocation (when the invocation's return is not void), the state of the receiver object, and the states of argument objects (when the invocation can modify the states of the argument objects). Automatic test-generation tools often do not create assertions but rely on uncaught exceptions or program crashes to detect problems in a program [11, 12].

To address insufficient test oracles of an automatically generated test suite, we have developed an automatic tool, called Orstra, to augment the test suite for guarding against regression faults. Orstra executes tests in the test suite and collects the class under test's object states exercised by the test suite; an object's state is characterized by the values of the object's transitively reachable fields [43]. On collected object states, Orstra invokes observers (public methods with non-void returns) of the class under test, collects their actual return values, and creates assertions for checking the returns of observers against their actual collected values. In addition, for each collected object state S, Orstra determines whether there is another collected object state S' that is *equivalent* to S (state equivalence is defined by graph isomorphism [8, 43]); if so, Orstra reconstructs S' with method sequences and creates an assertion for checking the state equivalence of S and S'.

This paper makes the following main contributions:

- We propose a framework for asserting the behavior of a method invocation in an object-oriented unit-test suite.
- We develop an automatic test-oracle-augmentation tool that systematically adds assertions into an automatically generated test suite in order to improve its capability of guarding against regression faults.
- We evaluate our approach on augmenting automatically generated tests for eleven Java classes taken from a variety of sources. The experimental results show that our test-oracle augmentation can effectively improve the fault-detection capability of a test suite.

The rest of this paper is organized as follows. Section 2 presents an illustrating example. Section 3 presents our framework for asserting behavior of a method invocation in a test suite. Section 4 presents our Orstra tool for automatically augmenting a

test suite. Section 5 presents an experiment to assess our approach. Section 6 discusses issues of the approach. Section 7 reviews related work, and Section 8 concludes.

2 Example

We next illustrate how Orstra augments an automatically generated test suite's regression oracle checking. As an illustrating example, we use a Java implementation of a bounded stack that stores unique elements. Stotts et al. [40] used this Java implementation to experiment with their algebraic-specification-based approach for systematically creating unit tests. In the abbreviated implementation shown in Figure 1, the class `MyInput` is the comparable type of elements stored in the stack. In the class implementation of the bounded stack, the array `elems` contains the elements of the stack, and `numberOfElements` is the number of the elements and the index of the first free location in the stack. The `max` is the capacity of the stack. The public methods in the class interface include two standard stack operations: `push` and `pop`, as well as five observer methods, whose returns are not `void`.

Given a Java class, existing automatic test-generation tools [11, 12, 31, 43, 44] can generate a test suite automatically for the class. For example, Jtest [31] allows users to set the length of calling sequences between one and three, and then generates random calling sequences whose lengths are not greater than the user-specified one. JCrasher [11] automatically constructs method sequences to generate non-primitive arguments and uses default data values for primitive arguments. JCrasher generates tests as calling sequences with the length of one.

For example, given the `UBStack` class, existing automatic test-generation tools [11, 12, 31, 43, 44] can generate test suites such as the example *test suite* `UBStackTest` with two tests (exported in the JUnit testing framework [19]) shown in Figure 2. Each *test* has several method sequences on the objects of the class. For example, `test1` creates a stack `s1` and invokes `push`, `top`, `pop`, and `isMember` on it in a row.

Note that there are no assertions generated in the `UBStackTest` test suite. Therefore, when the test suite is run, tools such as JCrasher [11] and CnC [12] detect problems by observing whether uncaught exceptions are thrown; tools such as Korat [8] detect problems by observing whether the execution of the test suite violates design-by-contract annotations [9, 23, 28] (equipped with the program under test), which are translated into run-time assertions [2, 36].

Given a test suite such as `UBStackTest`, Orstra systematically augments the test suite to produce an augmented test suite such as `UBStackAugTest` shown in Figure 3. For illustration, we annotate `UBStackAugTest` with line numbers and mark in bold font those lines of statements that correspond to the statements in `UBStackTest`. The augmented test suite `UBStackAugTest` is equipped with comprehensive assertions, which reflect the behavior of the current program version under test. These new assertions can guard against regression faults introduced in future program versions.

We next illustrate how Orstra automatically creates assertions for `UBStackTest` to produce `UBStackAugTest`. By running `UBStackTest`, Orstra dynamically monitors the method sequences executed by `UBStackTest` and collects the exercised state of a `UBStack`-receiver object by collecting the values of the re-


```
public class MyInput implements Comparable {
    private int o;
    public MyInput(int i) { o = i; }
    public boolean equals(Object that) {
       if (!(that instanceof MyInput)) return false;
       return (o == ((MyInput)that).o);
    }
}

public class UBStack {
    private Comparable[] elems;
    private int numberOfElements;
    private int max;
    public UBStack() { ... }
    //standard stack operations
    public void push(Comparable i) { ... }
    public void pop() { ... }
    //stack observer methods
    public int getNumberOfElements() { ... }
    public boolean isFull() { ... }
    public boolean isEmpty() { ... }
    public boolean isMember(Comparable i) {... }
    public MyInput top() { ... }
}
```

Fig. 1. A bounded stack implementation (`UBStack`) in Java

```
public class UBStackTest extends TestCase {
    public void test1() {
        UBStack s1 = new UBStack();
        MyInput i1 = new MyInput(3);
        s1.push(i1);
        s1.top();
        s1.pop();
        s2.isMember(i1);
    }

    public void test2() {
        UBStack s2 = new UBStack();
        s2.isEmpty();
        s2.isFull();
        s2.getNumberOfElements();
    }
}
```

Fig. 2. An automatically generated test suite `UBStackTest` for `UBStack`

ceiver object's transitively reachable fields. Based on the collected method invocations, Orstra identifies `UBStack`'s observer methods that are invoked by `UBStackTest`: `top()`, `isMember(new MyInput(3))`, `isEmtpy()`, `isFull()`, and `getNumberOfElements()`.

Then on each `UBStack`-receiver-object state exercised by `UBStackTest`, Orstra invokes the collected observer methods. For example, after the constructor invocation (shown in Line 2 of Figure 3), Orstra invokes the five observer methods on the `UBStack` object `s1`. After invoking these observer methods, Orstra collects their return values and then makes an assertion for each observer method by adding a JUnit assertion method (`assertEquals`), whose first argument is the observer method's return and second argument is the collected return value. The five inserted assertions are shown in Lines 4-9. Similarly, Orstra inserts assertions after the `push` invocation (shown in

```
0 public class UBStackAugTest extends TestCase {
1  public void testAug1() {
2   UBStack s1 = new UBStack();
3   //start inserting new assertions for observers
4   assertEquals(s1.isEmpty(), true);
5   assertEquals(s1.isFull(), false);
6   assertEquals(s1.getNumberOfElements(), 0);
7   MyInput temp_i1 = new MyInput(3);
8   assertEquals(s1.isMember(temp_i1), false);
9   assertEquals(s1.top(), null);
10  //finish inserting new assertions for observers
11  MyInput i1 = new MyInput(3);
12  s1.push(i1);
13  //start inserting new assertions for observers
14  assertEquals(s1.isEmpty(), false);
15  assertEquals(s1.isFull(), false);
16  assertEquals(s1.getNumberOfElements(), 1);
17  assertEquals(s1.isMember(temp_i1), true);
18  //finish inserting new assertions for observers
19  assertEquals(Runtime.genStateStr(s1.top()), "o:3;");
20  //insert no new assertions for top
21  s1.pop();
22  //start inserting new assertions for state equivalence
23  UBStack temp_s1 = new UBStack();
24  EqualsBuilder.reflectionEquals(s1, temp_s1);
25  //finish inserting new assertions for state equivalence
26  assertEquals(s2.isMember(i1), false);
27  //insert no new assertions for isMember
28 }
29
30 public void testAug2() {
31  UBStack s2 = new UBStack();
32  //insert no new assertions because the equivalent state
33  //has been asserted in test1
34  assertEquals(s2.isEmpty(), true);
35  assertEquals(s2.isFull(), false);
36  assertEquals(s2.getNumberOfElements(), 0);
37 }
39}
```

Fig. 3. An Orstra-augmented test suite for `UBStackTest`

Line 12) for asserting the state of the receiver `s1`. Because in `test1` of `UBStackTest`, there is an observer method `top` invoked immediately after the `push` invocation, in the inserted assertions for `s1` after the `push` invocation, Orstra does not include another duplicate `top` observer invocation. Then Orstra still adds an assertion for the original `top` invocation (shown in Line 19). When Orstra collects the return value of `top`, it determines that the value is not of a primitive type but of the `MyInput` type. It then invokes its own runtime helper method (**`Runtime`**`.genStateStr`) to collect the state-representation string of the `MyInput`-type return value. The string consists of the values of all transitively reachable fields of the `MyInput`-type object, represented as "`o:3;`", where `o` is the field name and `3` is the field value.

After the `top` invocation (shown in Line 19), Orstra inserts no new assertion for asserting the state of `s1` immediately after the `top` invocation, because Orstra dynamically determines `top` to be a state-preserving or side-effect-free method: all its invocations in the test suite do not modify the state of the receiver object.

After the `pop` invocation (shown in Line 21), Orstra detects that `s1`'s state is equivalent to another collected object state that is produced by a shorter method sequence: an

object state produced after the constructor invocation; Orstra determines state equivalence of two objects by comparing their state-representation strings. Therefore, instead of invoking observer methods on `s1`, Orstra constructs an assertion for asserting that the state of `s1` is equivalent to the state of `temp_s1`, which is produced after the constructor is invoked. Orstra creates the assertion by using an equals-assertion-builder method (**`EqualsBuilder`**`.reflectionEquals`) from the Apache Jakarta Commons subproject [4]. This method uses Java reflection mechanisms [5] to determine if two objects are equal based on field-by-field comparison. If an `equals` method is defined as a public method of the class under test, Orstra can also alternatively use the `equals` method for building the assertion.

After the `isMember` invocation (shown in Line 26), Orstra inserts no new assertion for asserting the state of `s1` immediately after the `isMember` invocation, because Orstra dynamically determines `isMember` to be a state-preserving method.

When augmenting `test2`, Orstra does not insert assertions for the state of `s2` immediately after the constructor invocation, because the object state that is produced by the same method sequence has been asserted in `testAug1`. In `testAug2`, Orstra adds assertions only for those observer-method invocations that are originally in `test2` (shown in Lines 34-36).

3 Framework

This section formalizes some notions introduced informally in the previous section. We first describe approaches for representing states of non-primitive-type objects and then compare these approaches. We finally describe how these state representations can be used to build assertions for the receiver object and return value of a method invocation.

3.1 State Representation

When a variable (such as the return of a method invocation) is of a primitive type or a primitive-object type such as `String` and `Integer`, Orstra asserts its value by comparing it with an expected value. When a variable (such as the return or receiver of a method invocation) is a non-primitive-type object, Orstra constructs assertions by using several types of state representations: method-sequence representation [43], concrete-state representation [43], and observer-abstraction representation [46].

Method-Sequence Representation The method-sequence-representation technique [43] represents the state of an object by using sequences of method invocations that produce the object (following Henkel and Diwan [22] who use the representation in mapping Java classes to algebras). Then Orstra can reconstruct or clone an object state by re-executing the method invocations in the method-sequence representation; the capability of reconstructing an object state is crucial when Orstra wants to assert that the state of the object under consideration is equivalent to that of another object constructed elsewhere.

The state representation uses symbolic expressions with the grammar shown below:

exp ::= prim | invoc "`.state`" | invoc "`.retval`"
args ::= ϵ | exp | args "`,`" exp
invoc ::= method "`(`" args "`)`"
prim ::= "`null`" | "`true`" | "`false`" | "`0`" | "`1`" | "`-1`" | ...

Each object or value is represented with an expression. Arguments for a method invocation are represented as sequences of zero or more expressions (separated by commas); the receiver of a non-static, non-constructor method invocation is treated as the first method argument. A static method invocation or constructor invocation does not have a receiver. The `.state` and `.retval` expressions denote the state of the receiver after the invocation and the return of the invocation, respectively. For brevity, the grammar shown above does not specify types for the expressions. A method is represented uniquely by its defining class, name, and the entire signature. (For brevity, we do not show a method's defining class or signature in the state-representation examples of this paper.) For example, in `test1`, the state of the object `s1` after the `push` invocation is represented by

```
push(UBStack<init>().state, MyInput<init>(3).state).state.
```

where `UBStack<init>` and `MyInput<init>` represent constructor invocations.

Note that the state representation based on method sequences allows tests to contain loops, arithmetic, aliasing, and polymorphism. Consider the following two tests `test3` and `test4`:

```
public void test3() {
  UBStack t = new UBStack();
  UBStack s3 = t;
  for (int i = 0; i <= 1; i++)
     s3.push(new MyInput(i));
}

public void test4() {
  UBStack s4 = new UBStack();
  int i = 0;
  s4.push(new MyInput(i));
  s4.push(new MyInput(i + 1));
}
```

Orstra dynamically monitors the invocations of the methods on the actual objects created at runtime and collects the actual argument values for these invocations. For example, it represents the states of both `s3` and `s4` at the end of `test3` and `test4` as `push(push(UBStack<init>().state, MyInput<init>(0)).state, MyInput<init>(1)).state`.

The above-shown grammar does not capture a method execution's side effect on an argument: a method can modify the state of a non-primitive-type argument and this argument can be used for another later method invocation. Following Henkel and Diwan's suggested extension [22], we can enhance the first grammar rule to address this issue:

exp ::= prim | invoc "`.state`" | invoc "`.retval`" | invoc "`.`arg_i"

where the added expression (invoc "`.`arg_i") denotes the state of the modified ith argument after the method invocation.

If test code modifies directly some public fields of an object without invoking any of its methods, these side effects on the object are not captured by method sequences

in the method-sequence representation. To address this issue, Orstra can be extended to create a public field-writing method for each public field of the object, and then monitor object-field accesses in the test code. If Orstra detects at runtime the execution of the object's field-write instruction in the test code, it can insert a corresponding field-writing method invocation in the method-sequence representation.

Concrete-State Representation A program is executed upon the program state that includes a program heap. The concrete-state representation of an object [43] considers only parts of the heap that are reachable from the object. We also call each part a "heap" and view it as a graph: nodes represent objects and edges represent fields. Let P be the set consisting of all primitive values, including `null`, integers, etc. Let O be a set of objects whose fields form a set F. (Each object has a field that represents its class, and array elements are considered index-labelled object fields.)

Definition 1. *A* heap *is an edge-labelled graph* $\langle O, E\rangle$*, where* $E = \{\langle o, f, o'\rangle | o \in O, f \in F, o' \in O \cup P\}$.

Heap isomorphism is defined as graph isomorphism based on node bijection [8].

Definition 2. *Two heaps* $\langle O_1, E_1\rangle$ *and* $\langle O_2, E_2\rangle$ *are* isomorphic *iff there is a bijection* $\rho : O_1 \rightarrow O_2$ *such that:*

$$E_2 = \{\langle \rho(o), f, \rho(o')\rangle | \langle o, f, o'\rangle \in E_1, o' \in O_1\} \cup \{\langle \rho(o), f, o'\rangle | \langle o, f, o'\rangle \in E_1, o' \in P\}.$$

The definition allows only object identities to vary: two isomorphic heaps have the same fields for all objects and the same values for all primitive fields.

The state of an object is represented with a *rooted* heap, instead of the whole program heap.

Definition 3. *A rooted heap is a pair* $\langle r, h\rangle$ *of a root object* r *and a heap* h *whose all nodes are reachable from* r.

Orstra linearizes rooted heaps into strings such that checking heap isomorphism corresponds to checking string equality. Figure 4 shows the pseudo-code of the linearization algorithm. The linearization algorithm traverses the entire rooted heap in the depth-first order, starting from the root. When the algorithm visits a node for the first time, it assigns a unique identifier to the node, and keeps this mapping in `ids` so that already assigned identifiers can be reused by nodes that appear in cycles. We can show that the linearization normalizes rooted heaps into strings. The states of two objects are *equivalent* if their strings resulted from linearization are the same.

Observer-Abstraction Representation The observer abstraction technique [46] represents the state of an object by using abstraction functions that are constructed based on observers. We first define an observer following Henkel and Diwan's work [22] on specifying algebraic specifications for a class:

```
Map ids; // maps nodes into their unique ids
String linearize(Node root, Heap <O,E>) {
  ids = new Map();
  return lin("root", root, <O,E>);
}

String lin(String fieldName, Node root, Heap <O,E>) {
  if (ids.containsKey(root))
    return fieldName+":"+String.valueOf(ids.get(root))+";";
  int id = ids.size() + 1;
  ids.put(root, id);
  StringBuffer rep = new StringBuffer();
  rep.append(fieldName+":"+String.valueOf(id)+";");
  Edge[] fields = sortByField({ <root, f, o> in E });
  foreach (<root, f, o> in fields) {
    if (isPrimitive(o))
      rep.append(f+":"+String.valueOf(o)+";");
    else
      rep.append(lin(f, o, <O,E>));
  }
  return rep.toString();
}
```

Fig. 4. Pseudo-code of the linearization algorithm

Definition 4. *An* observer *of a class c is a method ob in c's interface such that the return type of ob is not void.*

An observer invocation is a method invocation whose method is an observer. Given an object o of class c and a set of observer calls $OB = \{ob_1, ob_2, ..., ob_n\}$[1] of c, the observer abstraction technique represents the state of o with n values $OBR = \{obr_1, obr_2, ..., obr_n\}$, where each value obr_i represents the return value of observer call ob_i invoked on o.

When behavior of an object is to be asserted, Orstra can assert the observer-abstraction representation of the object: asserting the return values of observer invocations on the object.

Among different user-defined observers for a class, `toString()` [41] deserves special attention. This observer returns a string representation of the object, often being concise and human-readable. `java.lang.Object` [41] defines a default `toString`, which returns the name of the object's class followed by the unsigned hexadecimal representation of the hash code of the object. The Java API documentation [41] recommends developers to override this `toString` method in their own classes.

Comparison In this section, we compare different state representations in terms of their relationships and the extent of revealing implementation details, as well as their effects on asserting method invocation behavior.

We first define subsumption relationships among state representations as follows. State representation S_1 *subsumes* state representation S_2 if and only if any two objects that have the same S_1 representations also have the same S_2 representations. State representation S_1 *strictly subsumes* state representation S_2 if S_1 subsumes S_2 and for some objects `O` and `O'`, the S_1 representations differ but the S_2 representations do not. State representations S_1 and S_2 are *incomparable* if neither S_1 subsumes S_2 nor S_2 subsumes S_1. State representations S_1 and S_2 are *equivalent* if S_1 subsumes S_2 and S_2 subsumes S_1.

If state representation S_1 subsumes state representation S_2, and S_1 has been asserted (by checking whether the actual state representation is the same as the expected one), it is not necessary to assert S_2: asserting S_2 is *redundant* after we have asserted S_1.

The method-sequence representation strictly subsumes the concrete-state representation. The concrete-state representation strictly subsumes the observer-abstraction representation. Among different observers, the representation resulting from the `toString()` observer often subsumes the representation resulting from other observers and is often equivalent to the concrete-state representation.

Different state representations expose different levels of implementation details. If a state representation exposes more implementation details of a program, it is often more difficult for developers to determine whether the program behaves as expected once an assertion for the state representation is violated. In addition, If a state representation exposes more implementation details, developers can be overwhelmed by assertion violations that are not symptoms of regression faults but due to expected implementation changes (such as during program refactoring [18]). Although these assertion violations can be useful during software impact analysis [6], we prefer to put assertions on state representations that reveals fewer implementation details.

Among the three representations, the concrete-state representation exposes more implementation details than the other two representations: the concrete-state representation of an object is sensitive to changes on the object's field structure or the semantic of its fields, even if these changes do not cause any behavioral difference in the object's interface. To address this issue of the concrete-state representation, when Orstra creates an assertion for an object's concrete-state representation, instead of directly asserting the concrete-state representation string, Orstra asserts that the object is equivalent to another object produced with a different method sequence if such an object can be found (note that state equivalence is still determined based on the comparison of representation strings). This strategy is inspired by state-equivalence checking in algebraic-specifications-based testing [16, 22]. One such example is in Line 24 of Figure 3.

3.2 Method-Execution-Behavior Assertions

The execution of a test case produces a sequence of method executions.

Definition 5. *A* method execution *is a sextuple e = (m, S_{args}, S_{entry}, S_{exit}, $S_{args'}$, r) where m, S_{args}, S_{entry}, S_{exit}, $S_{args'}$, and r are the method name (including the signature), the argument-object states at the method entry, the receiver-object state at the method entry, the receiver-object state at the method exit, the argument-object states at the method exit, and the method return value, respectively.*

Note that when m's return is $void$, r is $void$; when m is a static method, S_{entry} and S_{exit} are empty; when m is a constructor method, S_{entry} is empty.

When a method execution e is a public method of the class under test C and none of e's indirect or direct callers is a method of C, we call that e is invoked on the interface

[1] Orstra does not use an observer defined in `java.lang.Object` [41].

of C. For each such method execution e invoked on the interface of C, if S_{exit} is not empty, S_{exit} can be asserted by using the following ways:

- If another method sequence can be found to produce an object state S' that is expected to be equivalent to S_{exit}, an assertion is created to compare the state representations of S' and S_{exit}.
- If an observer method ob is defined by the class under test, an assertion is created to compare the return of an ob invocation on S_{exit} with the expected value (the ways of comparing return values are described below).

As is discussed in Section 3.1, we do not create an assertion that directly compares the concrete-state representation string of the receiver object with the expected string, because such an assertion is too sensitive to some internal implementation changes that may not affect the interface behavior.

If a method invocation is a state-preserving method, then asserting S_{exit} is not necessary; instead, the existing purity analysis techniques [37, 39] can be exploited to statically check its purity if its purity is to be asserted.

Similarly, we can assert $S_{args'}$ in the same way as asserting S_{exit}. If a method invocation does not modify argument objects' states, then asserting $S_{args'}$ is not necessary.

For each method execution e that is invoked on the interface of the class under test, if r is not $void$, its return value r can be asserted by using the following ways:

- If r is of a primitive type (including primitive-type objects such as `String` and `Integer`), an assertion is created to compare r with the expected primitive value.
- If r is of the class-under-test type (which is a non-primitive type), an assertion is created by using the above ways of asserting a receiver-object state S_{exit}.
- If r is of a non-primitive type R but not the class-under-test type,
  — if the observer method `toString` is defined by R, an assertion is created to compare the return of the `toString` invocation on r with the expected string value;
  — otherwise, an assertion is created to compare r's concrete-state representation string with the expected representation string value[2].

When a method execution throws an uncaught exception, we can add an assertion for asserting that the exception is to be thrown and it is not necessary to add other assertions for S_{exit}, $S_{args'}$, or r.

4 Automatic Test-Oracle Augmentation

The preceding section presents a framework for asserting the behavior exhibited by a method execution in a test suite. Although developers can manually write assertions based on the framework, it is tedious to write comprehensive assertions as specified by the framework. Some automatic test-generation tools such as JCrasher [11] do not generate any assertions and some tools such as Jtest [31] generate a limited number of assertions. In practice, the assertions in an automatically generated test suite are often insufficient to provide strong oracle checking. This section presents our Orstra tool that automatically adds new assertions into an automatically generated test suite based on the proposed framework. The automatic augmentation consists of two phases: state-capturing phase and assertion-building phase. In the state-capturing phase, Orstra dynamically collects object states exercised by the test suite and the method sequences that are needed to reproduce these object states. In the assertion-building phase, Orstra builds assertions that assert behavior of the collected object states and the returns of observer methods.

4.1 State-Capturing Phase

In the state-capturing phase, Orstra runs a given test suite T (in the form of a JUnit test class [19]) for the class under test C and dynamically rewrites the bytecodes of each class at class loading time (based on the Byte Code Engineering Library (BCEL) [13]).

Orstra rewrites the T class bytecodes to collect receiver object references, method names, method signatures, arguments, and returns at call sites of those method sequences that lead to C-object states or argument-object states for C's methods. Then Orstra can use the collected method call information to reconstruct the method sequence that leads to a particular C-object state or argument-object state. The reconstructed method sequence can be used in constructing assertions for C-object states in the assertion-building phase.

Orstra also rewrites the C class bytecodes in order to collect a C-object's concrete-state representations at the entry and exit of each method call invoked through the C-object's interface. Orstra uses Java reflection mechanisms [5] to recursively collect all the fields that are reachable from a C-object and uses the linearization algorithm (shown in Figure 4) to produce the object's state-representation string.

Additionally Orstra collects the set OM of observer-method invocations exercised by T. These observer-method invocations are used to inspect and assert behavior of an C-object state in the assertion-building phase.

4.2 Assertion-Building Phase

In the assertion-building phase, Orstra iterates through each C-object state o exercised by the initial test suite T. If o is equivalent to a nonempty set O of some other object states exercised by T, Orstra picks the object state o' in O that is produced by the shortest method sequence m'. Then Orstra creates an assertion for asserting state equivalence by using the techniques described in Section 3.2.

In particular, if an `equals` method is defined in C's interface, Orstra creates the following JUnit assertion method (`assertTrue`) [19] to check state equivalence after invoking the method sequence m' to produce o':

```
C o' = m';
assertTrue(o.equals(o'))
```

[2] Note that we do not intend to create another method sequence that produces an object state that is expected to be equivalent to r but directly assert r's concrete-state representation string, because r is not of the class-under-test type and its implementation details often remain relatively stable.

Note that m' needs to be replaced with the actual method sequence in the exported assertion code.

If no `equals` method is defined in C's interface, Orstra creates an assertion by using an equals-assertion-builder method (`EqualsBuilder.reflectionEquals`), which is from the Apache Jakarta Commons subproject [4]. This method uses Java reflection mechanisms [5] to determine if two objects are equal by comparing their transitively reachable fields. We can show that if two objects o and o' have the same state representation strings, the return value of `EqualsBuilder.reflectionEquals(o, o')` is `true`. Orstra creates the following assertion to check state equivalence after invoking the method sequence m' to produce o':

```
C o' = m';
EqualsBuilder.reflectionEquals(o, o')
```

If o is not equivalent to any other object state exercised by T, Orstra invokes on o each observer method om in OM collected in the state-capturing phase. Orstra collects the return value r of the om invocation and makes an assertion by using the techniques described in Section 3.2.

In particular, if r is of a primitive type, Orstra creates the following assertion to check the return of om:

```
assertEquals(o.om, r_str);
```

where r_str is the string representation of r's value.

If r is of the C type, Orstra uses the above-described technique for constructing an assertion for a C object if there exist any other object states that are equivalent to r.

If r is of a non-primitive type R but not the C type, Orstra creates the following assertion if a `toString` method is defined in R's interface:

```
assertEquals((o.om).toString(), t_str);
```

where t_str is the return value of the `toString` method invocation. If no `toString` method is defined in R's interface, Orstra creates the following assertion:

```
assertEquals(Runtime.genStateStr(o.om), s_str);
```

where `Runtime.genStateStr` is Orstra's own runtime helper method for returning the concrete-representation string of an object state, and s_str is the concrete-state representation string of r.

The preceding assertion building techniques are generally exhaustive, enumerating possible mechanisms that developers may use to write assertions manually for these different cases.

In the end of the assertion-building phase, Orstra produces an augmented test suite, which is an exported JUnit test suite, including generated assertions together with the original tests in T.

Note that an automatically generated test suite can include a high percentage of redundant tests [43], which generally do not add value to the test suite. It is not necessary to run these redundant tests or add assertions for these redundant tests. To produce a compact test suite with necessary assertions, the implementation of Orstra actually first collects all nonequivalent method executions and creates assertions only for these method executions; therefore, the tests in the actually exported JUnit test suite may not correspond one-on-one to the tests in the original JUnit test suite.

Table 1. Experimental subjects

class	meths	public meths	ncnb loc	Jtest tests	JCrasher tests	faults
IntStack	5	5	44	94	6	83
UBStack	11	11	106	1423	14	305
ShoppingCart	9	8	70	470	31	120
BankAccount	7	7	34	519	135	42
BinSearchTree	13	8	246	277	56	309
BinomialHeap	22	17	535	6205	438	310
DisjSet	10	7	166	779	64	307
FibonacciHeap	24	14	468	3743	150	311
HashMap	27	19	597	5186	47	305
LinkedList	38	32	398	3028	86	298
TreeMap	61	25	949	931	1000	311

5 Experiment

This section presents our experiment conducted to address the following research question:

– RQ: Can our Orstra test-oracle-augmentation tool improve the fault-detection capability (which approximates the regression-fault-detection capability) of an automatically generated test suite?

5.1 Experimental Subjects

Table 1 lists eleven Java classes that we use in the experiment. These classes were previously used in evaluating our previous work [43] on detecting redundant tests. `UBStack` is the illustrating example taken from the experimental subjects used by Stotts et al. [40]. `IntStack` was used by Henkel and Diwan [22] in illustrating their approach of discovering algebraic specifications. `ShoppingCart` is an example for JUnit [10]. `BankAccount` is an example distributed with Jtest [31]. The remaining seven classes are data structures previously used to evaluate Korat [8]. The first four columns show the class name, the number of methods, the number of public methods, and the number of non-comment, non-blank lines of code for each subject.

To address the research question, our experiment requires automatically generated test suites for these subjects so that Orstra can augment these test suites. We then use two third-party test-generation tools, Jtest [31] and JCrasher [11], to automatically generate test inputs for these eleven Java classes. Jtest allows users to set the length of calling sequences between one and three; we set it to three, and Jtest first generates all calling sequences of length one, then those of length two, and finally those of length three. JCrasher automatically constructs method sequences to generate non-primitive arguments and uses default data values for primitive arguments. JCrasher generates

tests as calling sequences with the length of one. The fifth and sixth columns of Table 1 show the number of tests generated by Jtest and JCrasher.

Although our ultimate research question is to investigate how much better an augmented test suite guards against regression faults, we cannot collect sufficient real regression faults for the experimental subjects. Instead, in the experiment, we use general fault-detection capability of a test suite to approximate regression-fault-detection capability. In particular, we measure the fault-detection capability of a test suite before and after Orstra's augmentation. Then our experiment requires faults for these eleven Java classes. These Java classes were not equipped with such faults; therefore, we used Ferastrau [24], a Java mutation testing tool, to seed faults in these classes. Ferastrau modifies a single line of code in an original version in order to produce a faulty version. We configured Ferastrau to produce around 300 faulty versions for each class. For three relatively small classes, Ferastrau generates a much smaller number of faulty versions than 300. The last column of Table 1 shows the number of faulty versions generated by Ferastrau.

5.2 Measures

To measure the fault-detection capability of a test suite, we use a metric, *fault-exposure ratio* (FE): the number of faults detected by the test suite divided by the number of total faults. A higher fault-exposure ratio indicates a better fault-detection capability. The JUnit testing framework [19] reports that a test fails when an assertion in the test is violated or an uncaught exception is thrown from the test. An initial test suite generated by JCrasher or Jtest may include some failing tests when being run on the original versions of some Java classes shown in Table 1, because some automatically generated tests may be illegal, violating (undocumented) preconditions of some Java classes. Therefore, we determine that a test suite exposes the seeded fault in a faulty version if the number of failing tests reported on the faulty version is larger than the number of failing tests on the original version. We measure the fault-exposure ratio FE_{orig} of an initial test suite and the fault-exposure ratio FE_{aug} of its augmented test suite. We then measure the *improvement factor*, given by the equation: $\frac{FE_{aug}-FE_{orig}}{FE_{orig}}$. A higher improvement factor indicates a more substantial improvement of the fault-detection capability.

5.3 Experimental Results

Table 2 shows the experimental results. The results for JCrasher-generated test suites are shown in Columns 2-4 and the results for Jtest-generated test suites are shown in Columns 5-7. Columns 2 and 5 show the fault-exposure ratios of the original test suites (before test-oracle augmentation). Columns 3 and 6 show the fault-exposure ratios of the test suites augmented by Orstra. Columns 4 and 7 show the improvement factors of the augmented test suites over the original test suites. The last two rows show the average and median data for Columns 2-7.

Without containing any assertion, a JCrasher-generated test exposes a fault if an uncaught exception is thrown during the execution of the test. We observed that JCrasher-generated tests has 0% fault-exposure ratios for two classes (`ShoppingCart` and

Table 2. Fault-exposure ratios of Jtest-generated, JCrasher-generated, and augmented test suites, and improvement factors of test augmentation.

class	JCrasher-gen tests			Jtest-gen tests		
	orig	aug	improve	orig	aug	improve
IntStack	9%	40%	3.36	47%	47%	0.00
UBStack	39%	53%	0.36	60%	60%	0.00
ShoppingCart	0%	48%	∞	56%	56%	0.00
BankAccount	0%	98%	∞	98%	98%	0.00
BinSearchTree	8%	20%	1.58	20%	27%	0.34
BinomialHeap	18%	95%	4.19	85%	95%	0.12
DisjSet	23%	31%	0.36	26%	43%	0.65
FibonacciHeap	9%	96%	9.28	55%	96%	0.74
HashMap	14%	76%	4.30	22%	76%	2.43
LinkedList	7%	35%	3.73	45%	45%	0.01
TreeMap	2%	89%	54.40	12%	89%	6.29
Average	12%	62%	9.06	48%	67%	0.96
Median	9%	53%	3.55	47%	60%	0.12

`BankAccount`), because no seeded faults for these two classes cause uncaught exceptions. Jtest equips its generated tests with some assertions: these assertions typically assert those method invocations whose return values are of primitive types. (Section 7 discusses main differences between Orstra and Jtest's assertion creation.) Generally, Jtest-generated test suites have higher fault-exposure ratios than JCrasher-generated test suites. The phenomenon is due to two factors: Jtest generates more test inputs (with longer method sequences) than JCrasher, and Jtest has stronger oracle checking (with additional assertions) than JCrasher.

After Orstra augments the JCrasher-generated test suites with additional assertions, we observed that the augmented test suites achieve substantial improvements of fault-exposure ratios. After augmenting the JCrasher-generated test suite for `TreeMap`, Orstra achieves an improvement factor of even beyond 50. The augmented Jtest-generated test suites also gain improvements of fault-exposure ratios (although not substantially as JCrasher-generated test suites), except for the first four classes. These four classes are relatively simple and seeded faults for these classes can be exposed with a less comprehensive set of assertions; Jtest-generated assertions are already sufficient to expose those exposable seeded faults.

5.4 Threats to Validity

The threats to external validity primarily include the degree to which the subject programs and their existing test suites are representative of true practice. Our subjects are from various sources and the Korat data structures have nontrivial size for unit testing. Our experiment had used initial test suites automatically generated by two third-party tools, one of which (Jtest) is popular and used in industry. These threats could be further reduced by experiments on more subjects and third-party tools. The main threats

to internal validity include instrumentation effects that can bias our results. Faults in our tool implementation, Jtest, or JCrasher might cause such effects. To reduce these threats, we have manually inspected the source code of augmented tests and execution traces for several program subjects. The main threats to construct validity include the uses of those measurements in our experiment to assess our tool. To assess the effectiveness of our test-oracle-augmentation tool, we measure the exposure ratios of faults seeded by a mutation testing tool to approximate the exposure ratios of real regression faults introduced as an effect of changes made in the maintenance process. Although empirical studies showed that faults seeded by mutation testing tools yield trustworthy results [3], these threats can be reduced by conducting more experiments on real regression faults.

6 Discussion

6.1 Analysis Cost

In general, the number of assertions generated for an initial test suite can be approximately characterized as

$$|assertions| = O(|nonEqvStates| \times |observers| + |statesEqvToAnother|)$$

where $|nonEqvStates| \times |observers|$ is the number of nonequivalent object states exercised by the initial test suite being multiplied by the number of observer calls exercised by the initial test suite; recall that Orstra generates an assertion for the return of an observer invoked on a nonequivalent object state. $|statesEqvToAnother|$ is the number of object states (produced by nonequivalent method executions in the initial test suite) that can be found to be equivalent to another object state produced by a different method sequence; recall that Orstra generates an assertion for asserting that an object state produced by a method sequence is equivalent to another object state produced by a different method sequence if any.

Using Orstra in regression testing activities incurs two types of extra cost. The first type is the cost of augmenting the initial test suite. In our experiment, the elapsed real time of running our test augmentation is reasonable, being up to several seconds, determined primarily by the class complexity, the number of tests in the test suite, the number of generated assertions. Note that Orstra needs to be run once when the initial test suite is augmented for the first time, and later to be run when reported assertion violations are determined not to be caused by regression faults. In future work, following the idea of repairing GUI regression tests [27], we plan to improve Orstra so that it can fix those violated assertions in the augmented test suite without re-augmenting the whole initial test suite.

The second type of cost is the cost of running additional assertion checking in the augmented test suite, determined primarily by the number of generated assertions. Although this cost is incurred every time the augmented test suite is run (after the program is changed), running the initial unit-test suite is often fast and running these additional assertion checking slows down the execution of the test suite within several factors. Indeed, if an initial test suite exercises many non-equivalent object states and the program under test has many observer methods, the cost of both augmenting the test suite and running the augmented test suite could be high. Under these situations, developers can configure Orstra to trade weaker oracle checking for efficiency by invoking a subset of observer methods during assertion generation. In addition, regression test prioritization [15] or test selection [20] for Java programs can be used to order or select tests in the Orstra-augmented test suite for execution when the execution time is too long.

6.2 Fault-Free Behavioral Changes

Orstra observes behavior of the program under test when being exercised by a test suite and then automatically adds assertions to the test suite to assert the program behavior is preserved after future program changes. Indeed, sometimes violations of inserted assertions do not necessarily indicate real regression faults. For example, consider that the program under test contains a fault, which is not exposed by the initial test suite. Orstra runs the test suite on the current (faulty) version and create assertions, some of which assert wrong behavior. Later developers find the fault and fix the program. When running the Orstra-augmented test suite on the new program version, assertion violations are reported but there are no regression faults. In addition, although Orstra has been carefully designed to assert as few implementation details in object-state representation as possible, some program changes may violate inserted assertions but still preserve program behavior that developers care about. To help developers to determine whether an assertion violation in an augmented test suite indicates real regression faults, we can use change impact analysis tools such as Chianti [33] to identify a set of affecting changes that were responsible for the assertion violation.

Some types of programs (such as multi-threaded programs or programs whose behaviors are related to time) may exhibit nondeterministic or different behaviors across multiple runs: running the same test suite twice may produce different observer returns or receiver-object states. For example, a `getTime` method returns the current time and a `getRandomNumber` method returns a random number. After we add assertions for these types of method returns in a test suite, running the augmented test suite on the current or new program version can report assertion violations, which do not indicate real faults or regression faults. To address this issue, we can run a test suite multiple times on the current program version and remove those assertions that are not consistently satisfied across multiple runs.

6.3 Availability of Observers

Orstra creates assertions for the returns of observers of the class under test. These observer calls may already exist in the initial test suite or may be invoked by Orstra to assert object-state behavior. Although observers are common in a class interface, there are situations where a class interface includes few or no observers. Even when a class interface includes no observer, we can still apply Orstra to augment a test suite generated for the class by asserting that a receiver-object state produced by a method sequence is equivalent to another receiver-object state produced by a different method sequence.

6.4 Iterations of Augmentation

Orstra runs an automatically generated test suite and then adds assertions to the test suite to produce an augmented test suite. When some observer methods are state-modifying methods, running them for preparing assertion checking in the augmented test suite can produce new receiver-object states that are not exercised by the initial test suite. Therefore, if we apply Orstra on the augmented test suite again, the second iteration of augmentation can produce a test suite with more assertion checking and thus often stronger oracle checking. However, if the augmented test suite after the first iteration does not produce any new receiver-object state, the second or later iteration of augmentation adds no new assertions to the test suite.

6.5 Quality of Automatically Generated Unit-Test Suites

The tests generated by JCrasher and Jtest (the two third-party test-generation tools used in the experiment) include a relatively high number of redundant tests [43], which do not contribute to achieving new structural coverage or better fault-detection capability. Rostra and Symstra (two test-generation tools developed in our previous work [43, 44]) can generate a test suite of higher quality (e.g., higher structural coverage) than a test suite generated by JCrasher or Jtest. Augmenting a test suite generated by Rostra or Symstra can achieve a higher improvement factor than augmenting a test suite generated by JCrasher or Jtest. In general, the higher quality a test suite is of, the higher improvement factor Orstra can achieve when augmenting the test suite.

6.6 Augmentation of Other Types of Test Suites

Although Orstra focuses on augmenting a unit-test suite, it is straightforward to extend Orstra to augment an integration-test suite, which intends to test the interactions of multiple classes. When we assert the return values of a method execution in an integration-test suite, we can directly apply Orstra without any modification. When we assert the receiver-object state at a method exit, we can adapt Orstra to invoke on the receiver object the observer methods of the receiver-object class rather than the observer methods of all the classes under test because there are multiple classes under test for an integration-test suite.

So far Orstra has been evaluated on augmenting an automatically generated test suite. Generally Orstra can also be used to augment a manually generated test suite, because the input to Orstra is simply a JUnit test class no matter whether it is generated automatically or manually. Because it is tedious to manually write comprehensive assertions for a test suite, a manually written test suite often does not have comprehensive assertions. We hypothesize that applying Orstra to augment a manually generated test suite can also improve the test suite's fault-detection capability. We plan to validate this hypothesis in our future experiments.

6.7 Incorporation of Oracle Augmentation in Test Generation

Orstra has been developed as an independent component that can augment any test suite in the form of a JUnit test class. Orstra can also be incorporated into the test-generation process of an existing test-generation tool as a two-step process. In the first step, the tool generates test inputs and runs these generated test inputs to collect method returns and object states. This step combines the existing test-generation process and Orstra's state capturing phase. The second step includes Orstra's assertion-building phase. Some existing test-generation tools such as JCrasher do not run generated test inputs during their test-generation process. Then these tools can loosely incorporate Orstra by adopting this two-step process. Some existing tools such as Jtest, Rostra [43], and Symstra [44] actually run generated test inputs during their test-generation process. Then these tools can tightly incorporate Orstra by including Orstra's state-capturing and assertion-building phases when these tools run the generated test inputs during the test-generation process. In fact, Orstra has been incorporated into Rostra and Symstra as an optional component for adding assertions to their generated tests.

7 Related Work

Richardson [34] developed the TAOS (Testing with Analysis and Oracle Support) toolkit, which provides different levels of test oracle support. For example, in lower levels, developers can write down expected outputs for a test input, specify ranges for variable values, or manually inspect actual outputs. The oracle support provided by our Orstra tool is in TAOS' lower levels: generating expected outputs for test inputs. In higher levels, developers can use specification languages (such as Graphical Interval Logic Langauge and Real-Time Interval Logic Language) to specify temporal properties. There exist a number of proposed approaches for providing oracle supports based on different types of specifications [9, 14, 26, 32, 35]. In particular, for testing Java programs, Cheon and Leavens [9] developed a runtime verification tool for Java Modelling Language (JML) [23] and then provided oracle supports for automatically generated tests. This oracle checking approach was also adopted by automatic specification-based test generation tools such as Korat [8]. Different from these specification-based oracle supports, Orstra does not require specifications but Orstra can enhance oracle checking only for exposing regression faults.

When specifications do not exist, automatic test-generation tools such as JCrasher [11] and CnC [12] use program crashes or uncaught exceptions as symptoms of the current program version's faulty behavior. Like Orstra, Jtest [31] can also create some assertions for its generated tests. Orstra differs from Jtest in several ways. Jtest creates assertions for its own generated tests only, whereas Orstra can augment any third-party test suite. Jtest creates assertions for method invocations whose return values are of primitive types, whereas Orstra creates more types of assertions, such as asserting returns with non-primitive types and asserting behavior of receiver-object states. Unlike Orstra, Jtest does not systematically or exhaustively create assertions to assert exercised program behavior. Our experimental results (shown in Section 5.3) indicate that Orstra can still effectively augment a Jtest-generated test suite, which has been equipped with Jtest-generated assertions.

Saff and Ernst [38] as well as Orso and Kennedy [29] developed techniques for capturing and replaying interactions between a selected subsystem (such as a class) and the rest of the application. Their techniques focus on creating fast, focused unit tests

from slow system-wide tests, whereas our Orstra tool focuses on adding more assertions to an existing unit-test suite. In addition, Orstra's techniques go beyond capturing and replaying, because Orstra creates new helper-method invocations for assertion checking and these new method invocations might not be exercised in the original test suite.

Memon et al. [25] model a GUI state in terms of the widgets that the GUI contains, their properties, and the values of the properties. Their experimental results show that comparing more-detailed GUI states (e.g., GUI states associated with all or visible windows) from two versions can detect faults more effectively than comparing less-detailed GUI states (e.g., GUI states associated with the active window or widget). Our experiment shows a similar result: checking more-detailed behavior (with augmented test suites) can more effectively expose regression faults.

Both Harrold et al's spectra comparison approach [21] and our previous value-spectra comparison approach [47] also focus on exposing regression faults. Program spectra usually capture internal program execution information and these approaches compare program spectra from two program versions in order to expose regression faults. Our new Orstra tool compares interface-visible behavior of two versions without comparing internal execution information. On one hand, Orstra may not report behavioral differences that are reported by spectra comparison approaches, if these internal behavioral differences cannot cause behavioral differences in the interface. On the other hand, Orstra may report behavioral differences that are not reported by spectra comparison approaches, if these behavioral differences are exhibited only by new Orstra-invoked observers (spectra comparison approaches do not create any new method invocation).

When there are no oracles for a large number of automatically generated tests, developers cannot afford to inspect the results of such a large number of tests. Our previous operational violation approach [45] selects a small subset of automatically generated tests for inspection; these selected tests violates the operational abstractions [17] inferred from the existing test suite. Pacheco and Ernst [30] extended the approach by additionally using heuristics to filter out illegal test inputs. Agitar Agitator [1] automatically generates initial tests, infers operational-abstraction-like observations, lets developers confirm these observations to assertions, and generates more tests to violate these inferred and confirmed observations. The operational violation approach primarily intends to expose faulty behavior exhibited by new generated tests on the current program version, whereas Orstra intends to enhance the oracle checking of an existing test suite so that it has an improved capability of exposing faulty behavior exhibited by the same test suite on future program versions.

Orstra has been implemented based on our two previous approaches. Our previous Rostra approach [43] provides state representation and comparison techniques, but Rostra compares states in order to detect redundant tests out of automatically generated tests. Our previous Obstra approach [46] also invokes observers on object states exercised by an existing test suite. Obstra uses the return values of observers to abstract concrete states and constructs abstract-object-state machines for inspection. Obstra allows developers to inspect the behavior of the current program version, whereas Orstra uses the return values of observers as well as receiver object states to assert that behavior of future program versions is the same as behavior of the current program version.

In contrast to Rostra and Obstra, Orstra makes new contributions in developing an approach for enhancing the regression oracle checking of an automatically generated test suite.

8 Conclusion

An automatic test-generation tool can be used to generate a large number of test inputs for the class under test, complementing manually generated tests. However, without specifications these automatically generated test inputs do not have test oracles to guard against faults in the current program version or regression faults in future program versions. We have developed a new automated approach for augmenting an automatically generated test suite in guarding against regression faults. In particular, we have proposed a framework for asserting behavior of a method invocation in an object-oriented unit-test suite. Based on the framework, we have developed an automatic test-oracle-augmentation tool, called Orstra, that systematically adds assertions into an automatically generated test suite in order to improve its capability of guarding against regression faults. We have conducted an experiment to assess the effectiveness of augmenting tests generated by two third-party test-generation tools. The results show that Orstra can effectively increase the fault-detection capability of automatically generated tests by augmenting their regression oracle checking.

Acknowledgments

We would like to thank Alex Orso and Andreas Zeller for discussions that lead to the work described in this paper. We thank Darko Marinov for providing the Ferastrau mutation testing tool and Korat subjects used in the experiment.

References

1. Agitar Agitatior 2.0, Novermber 2004. `http://www.agitar.com/`.
2. D. M. Andrews. Using executable assertions for testing and fault tolerance. In *Proc. the 9th International Symposium on Fault-Tolerant Computing*, pages 102–105, 1979.
3. J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proc. 27th International Conference on Software Engineering*, pages 402–411, 2005.
4. The Jakarta Commons Subproject, 2005. `http://jakarta.apache.org/commons/lang/apidocs/org/apache/commons/lang/builder/EqualsBuilder.html`.
5. K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language*. Addison-Wesley Longman Publishing Co., Inc., 2000.
6. R. S. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.
7. K. Beck. *Extreme programming explained*. Addison-Wesley, 2000.
8. C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *Proc. International Symposium on Software Testing and Analysis*, pages 123–133, 2002.

9. Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *Proc. 16th European Conference Object-Oriented Programming*, pages 231–255, June 2002.
10. M. Clark. Junit primer. Draft manuscript, October 2000.
11. C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34:1025–1050, 2004.
12. C. Csallner and Y. Smaragdakis. Check 'n' Crash: Combining static checking and testing. In *Proc. 27th International Conference on Software Engineering*, pages 422–431, May 2005.
13. M. Dahm and J. van Zyl. Byte Code Engineering Library, April 2003. `http://jakarta.apache.org/bcel/`.
14. L. K. Dillon and Y. S. Ramakrishna. Generating oracles from your favorite temporal logic specifications. In *Proc. 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 106–117, 1996.
15. H. Do, G. Rothermel, and A. Kinneer. Empirical studies of test case prioritization in a JUnit testing environment. In *Proc. 15th International Symposium on Software Reliability Engineering*, pages 113–124, 2004.
16. R.-K. Doong and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Trans. Softw. Eng. Methodol.*, 3(2):101–130, 1994.
17. M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Softw. Eng.*, 27(2):99–123, 2001.
18. M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.
19. E. Gamma and K. Beck. JUnit, 2003. `http://www.junit.org`.
20. M. J. Harrold, J. A. Jones, T. Li, D. Liang, and A. Gujarathi. Regression test selection for Java software. In *Proc. 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 312–326, 2001.
21. M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi. An empirical investigation of the relationship between spectra differences and regression faults. *Journal of Software Testing, Verification and Reliability*, 10(3):171–194, 2000.
22. J. Henkel and A. Diwan. Discovering algebraic specifications from Java classes. In *Proc. 17th European Conference on Object-Oriented Programming*, pages 431–456, 2003.
23. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report TR 98-06i, Department of Computer Science, Iowa State University, June 1998.
24. D. Marinov, A. Andoni, D. Daniliuc, S. Khurshid, and M. Rinard. An evaluation of exhaustive testing for data structures. Technical Report MIT-LCS-TR-921, MIT CSAIL, Cambridge, MA, September 2003.
25. A. M. Memon, I. Banerjee, and A. Nagarajan. What test oracle should I use for effective GUI testing? In *Proc. 18th IEEE International Conference on Automated Software Engineering*, pages 164–173, 2003.
26. A. M. Memon, M. E. Pollack, and M. L. Soffa. Automated test oracles for GUIs. In *Proc. 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 30–39, 2000.
27. A. M. Memon and M. L. Soffa. Regression testing of GUIs. In *Proc. 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 118–127, 2003.
28. B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.
29. A. Orso and B. Kennedy. Selective capture and replay of program executions. In *Proc. 3rd International ICSE Workshop on Dynamic Analysis*, pages 29–35, St. Louis, MO, May 2005.

30. C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *Proc. 19th European Conference on Object-Oriented Programming*, pages 504–527, Glasgow, Scotland, July 2005.
31. Parasoft Jtest manuals version 4.5. Online manual, April 2003. `http://www.parasoft.com/`.
32. D. Peters and D. L. Parnas. Generating a test oracle from program documentation. In *Proc. 1994 Internation Symposium on Software Testing and Analysis*, pages 58–65, 1994.
33. X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of Java programs. In *Proc. 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 432–448, 2004.
34. D. J. Richardson. TAOS: Testing with analysis and oracle support. In *Proc. 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 138–153, 1994.
35. D. J. Richardson, S. L. Aha, and T. O. O'Malley. Specification-based test oracles for reactive systems. In *Proc. 14th International Conference on Software Engineering*, pages 105–118, 1992.
36. D. S. Rosenblum. Towards a method of programming with assertions. In *Proc. 14th International Conference on Software Engineering*, pages 92–104, 1992.
37. A. Rountev. Precise identification of side-effect-free methods in Java. In *Proc. 20th IEEE International Conference on Software Maintenance*, pages 82–91, Sept. 2004.
38. D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic test factoring for Java. In *Proc. 21st IEEE International Conference on Automated Software Engineering*, pages 114–123, Long Beach, CA, November 2005.
39. A. Salcianu and M. Rinard. Purity and side effect analysis for Java programs. In *Proc. 6th International Conference on Verification, Model Checking and Abstract Interpretation*, pages 199–215, Paris, France, January 2005.
40. D. Stotts, M. Lindsey, and A. Antley. An informal formal method for systematic JUnit test case generation. In *Proc. 2002 XP/Agile Universe*, pages 131–143, 2002.
41. Sun Microsystems. Java 2 Platform, Standard Edition, v 1.4.2, API Specification. Online documentation, Nov. 2003. `http://java.sun.com/j2se/1.4.2/docs/api/`.
42. W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *Proc. 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 97–107, 2004.
43. T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *Proc. 19th IEEE International Conference on Automated Software Engineering*, pages 196–205, Sept. 2004.
44. T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proc. 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 365–381, April 2005.
45. T. Xie and D. Notkin. Tool-assisted unit test selection based on operational violations. In *Proc. 18th IEEE International Conference on Automated Software Engineering*, pages 40–48, 2003.
46. T. Xie and D. Notkin. Automatic extraction of object-oriented observer abstractions from unit-test executions. In *Proc. 6th International Conference on Formal Engineering Methods*, pages 290–305, Nov. 2004.
47. T. Xie and D. Notkin. Checking inside the black box: Regression testing by comparing value spectra. *IEEE Transactions on Software Engineering*, 31(10):869–883, October 2005.