Appears in *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE 2004)*, Linz, Austria.

Rostra: A Framework for Detecting Redundant Object-Oriented Unit Tests

Tao Xie[1] Darko Marinov[2] David Notkin[1]

[1] Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195, USA

[2] MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA 02139, USA

[1]{taoxie,notkin}@cs.washington.edu [2]marinov@lcs.mit.edu

Abstract

Object-oriented unit tests consist of sequences of method invocations. Behavior of an invocation depends on the state of the receiver object and method arguments at the beginning of the invocation. Existing tools for automatic generation of object-oriented test suites, such as Jtest and JCrasher for Java, typically ignore this state and thus generate redundant tests that exercise the same method behavior, which increases the testing time without increasing the ability to detect faults.

This paper proposes Rostra, a framework for detecting redundant unit tests, and presents five fully automatic techniques within this framework. We use Rostra to assess and minimize test suites generated by test-generation tools. We also present how Rostra can be added to these tools to avoid generation of redundant tests. We have implemented the five Rostra techniques and evaluated them on 11 subjects taken from a variety of sources. The experimental results show that Jtest and JCrasher generate a high percentage of redundant tests and that Rostra can remove these redundant tests without decreasing the quality of test suites.

1. Introduction

Several tools for automatic generation of object-oriented unit test suites, such as Jtest [21] (a commercial tool for Java) and JCrasher [7] (a research prototype for Java), test a class by generating sequences of method invocations. Each test consists of one sequence; when two sequences differ, these tools conservatively assume that the tests are not equivalent. However, there are many cases when different method sequences exercise the same behavior of the class under test. For example, two sequences can produce equivalent objects because some invocations do not modify state or because different state modifications result in the same state. Intuitively, invoking the same methods on such equivalent objects is redundant. Since testing is typically constrained by time limits, a key issue is to avoid producing and executing redundant tests that only increase the testing time and do not increase the ability to detect faults.

We propose Rostra, a novel framework for detecting redundant unit tests based on equivalent objects. Within Rostra, we present five techniques with different tradeoffs in 1) the assumptions about the code under test, 2) time and space taken to find redundant tests, and 3) the number of redundant tests found. These techniques are fully automatic and do not require any user input, except that two of the techniques assume that classes have properly implemented equality methods. Ideally, if a method claims that two objects are equal, they should be observationally equivalent [10, 17], i.e., have the same behavior for all method sequences they can be subject to. This is typically the case for `equals` methods in Java classes: the `java.lang.Object` class defines the `equals` method and subclasses often override it, as it is used pervasively, for example, to compare elements in the Java collections [27].

Some existing testing tools also consider equivalent objects, but differently than Rostra. For example, AsmLT [11] requires the user to provide an abstraction function [17] and defines objects to be equivalent if they map to the same abstract value. We can view some of our techniques as automatically defining an abstraction function based on `equals`. Several other projects [2, 10, 14] define equivalent objects using observational equivalence, but checking it precisely is expensive: by definition it takes infinite time (to check all method sequences), so in practice approximations are used. Our techniques take much less time and are more appropriate for testing.

We can use Rostra for several testing tasks. Rostra enables assessing the quality of a test suite in terms of non-equivalent objects and non-redundant tests, and we can thus compare the quality of different test suites. We can also select a subset of automatically generated tests to augment an existing (manually or automatically generated) test suite. We can further minimize an automatically generated test suite for manual inspection or regression testing. Finally, existing test-generation tools can incorporate Rostra into their test generation to avoid generation and execution of

redundant tests and instead invest the time in generation of non-redundant tests that exercise more method behaviors.

This paper makes the following contributions:

- We propose Rostra, a formal framework for detecting equivalent object states and redundant tests.
- We make Rostra concrete with five techniques and present an implementation of these techniques.
- We propose four practical applications of Rostra.
- We evaluate Rostra on 11 subjects taken from a variety of sources. The experimental results show that around 90% of the tests generated by Jtest for all subjects and 50% of the tests generated by JCrasher for almost half of the subjects are redundant. The results also show that removing these redundant tests does not decrease the branch coverage, exception coverage, and fault detection capability of the test suites.

2. Example

We next illustrate how our techniques determine redundant tests. As a running example, we use an integer stack implementation taken from Henkel and Diwan [14]. Figure 1 shows the relevant parts of the code.

The following is an example *test suite* with three tests for the `IntStack` class:

```
Test 1 (T1):
  IntStack s1 = new IntStack();
  s1.isEmpty();
  s1.push(3);
  s1.push(2);
  s1.pop();
  s1.push(5);

Test 2 (T2):
  IntStack s2 = new IntStack();
  s2.push(3);
  s2.push(5);

Test 3 (T3):
  IntStack s3 = new IntStack();
  s3.push(3);
  s3.push(2);
  s3.pop();
```

Each *test* has a method sequence on the objects of the class. For example, T2 creates a stack `s2` and invokes two `push` methods on it. Tests of this form are generated by tools such as Jtest [21] and JCrasher [7]. For such tests, the correctness checking typically relies on design-by-contract annotations [16, 20]; if the code has annotations, the tools translate them into run-time assertions [5, 21] that are checked during the execution. If there are no annotations, the tools only check the robustness of the code: they execute the tests and check for uncaught exceptions [7].

To determine redundant tests, our techniques dynamically monitor test executions. Each execution consists of transitions on the state of the Java program. Our techniques track these transitions at the granularity of methods:

```
public class IntStack {
  private int[] store;
  private int size;
  private static final int INITIAL_CAPACITY = 10;
  public IntStack() {
    this.store = new int[INITIAL_CAPACITY];
    this.size = 0;
  }
  public void push(int value) {
    if (this.size == this.store.length) {
      int[] store = new int[this.store.length * 2];
      System.arraycopy(this.store, 0, store, 0, this.size);
      this.store = store;
    }
    this.store[this.size++] = value;
  }
  public int pop() {
    return this.store[--this.size];
  }
  public boolean isEmpty() {
    return (this.size == 0);
  }
  public boolean equals(Object other) {
    if (!(other instanceof IntStack)) return false;
    IntStack s = (IntStack)other;
    if (this.size != s.size) return false;
    for (int i = 0; i < this.size; i++)
      if (this.store[i] != s.store[i]) return false;
    return true;
  }
}
```

Figure 1. An integer stack implementation

each test execution produces a sequence of method executions. Each *method execution* is characterized by the actual method that is invoked and a *representation* of the state (receiver object and method arguments) at the beginning of the execution. We call this state *method-entry state*, or simply state when it is clear from the context. For instance, T2 has three method executions:

1. a constructor without arguments is invoked;
2. `push` adds `3` to the empty stack;
3. `push` adds `5` to the previous stack.

In this list, we use English language to describe the method-entry states. The techniques that we compare use several formal representations for a state and several approaches for determining *equivalent* states (Section 3.2).

We call two method executions equivalent if they are invocations of the same method on equivalent states. Our framework considers *redundant* tests: a test is redundant for a test suite if every method execution of the test is equivalent to some method execution of some test from the suite (Section 3.4).

We next briefly explain different techniques for determining equivalent states and illustrate redundant tests that these techniques find in the example test suite.

WholeSeq: This is the most conservative technique that models the existing test-generation tools. Two tests are considered equivalent only if they are identical. The technique

represents states using method sequences that create objects and compares states using sequence equality. It finds all three example tests to be non-redundant.

ModifyingSeq: This technique improves on the previous one by using in state representation only those method invocations that actually modify the state. It finds that T3 is redundant, because it exercises a subset of method executions that T1 exercises.

WholeState: This technique uses the whole concrete state for representation and compares states by isomorphism (Section 3.2.2). It also finds that T3 is redundant because of T1. However, it does not find T2 to be redundant because of T1: these two tests have different concrete states before `push(5)`—the array `store` has the value `[3,0]` in `s2` and the value `[3,2]` in `s1`.

MonitorEquals: This technique leverages the `equals` method to extract only the relevant parts of the state. It finds T2, as well as T3, to be redundant because of T1. Although the whole concrete states in T2 and T1 before `push(5)` are different, the relevant parts of the states are the same, namely the subarray of `store` up to `size` is `[3]`.

PairwiseEquals: This technique uses directly the `equals` method to compare pairs of states. In the running example, it finds the same redundant tests as the previous technique.

3. Formal Framework

This section formalizes the notions introduced informally in the previous section. We first discuss the assumptions that our techniques make about the code under test. We next describe approaches for representing states and comparing them for equivalence. We then describe how each of the five techniques builds the appropriate representation and finds equivalent states. We finally show how equivalent states give rise to equivalent method executions and define redundant tests and test-suite minimization.

3.1. Assumptions

All five techniques make the following assumption about the code under test:

A1: Method executions are deterministic given the state reachable from the receiver and other arguments.

This is realistic for single-threaded code; otherwise, different executions for the same input may produce different results, so model-checking techniques are more applicable than testing.

Techniques based on method-sequence representation (Section 3.3.1 and 3.3.2) make an additional assumption:

A2s: Each method can only modify the state of the receiver and return a result.

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

Figure 2. Grammar for symbolic expressions

"State of the receiver" refers to the abstract state, not only the fields of the concrete `this` object. For example, if the receiver is a head of a linked list of nodes, any node can be modified, not only the head. Henkel and Diwan make the same assumption for algebraic specifications [14].

The WholeState technique makes no other assumption.

Techniques based on the user-defined `equals` methods (Section 3.3.4 and 3.3.5) make an additional assumption:

A2e: The `equals` methods are implemented to respect the contract in `java.lang.Object` [27].

The contract requires that each `equals` implements an equivalence relation, i.e., it should be reflexive, symmetric, and transitive. In practice, we have found most `equals` methods to implement observational equivalence [10]: if `equals` is stronger (i.e., returns `false` for some objects that are observationally equivalent), our techniques may not remove some ideally redundant tests; if `equals` is weaker (i.e., returns `true` for some objects that are not observationally equivalent), our techniques may remove some ideally non-redundant tests. Rostra can dynamically check an approximation of observational equivalence for `equals` and help the user tune the method.

3.2. State Representation and Comparison

Our techniques use two main approaches for state representation: 1) method sequences and 2) concrete states of the objects. Both approaches view classes under test as having a set of methods (represented uniquely by their defining class, name, and the entire signature) and consider constructors as methods.

3.2.1. Method Sequences Each execution of a test creates several objects and invokes methods on these objects. Our method-sequence approach represents states using sequences of method invocations, following Henkel and Diwan who use the representation to map Java classes to algebras [14]. The state representation uses symbolic expressions with the concrete grammar shown in Figure 2. Each object and value are represented with an expression. Arguments for method invocations are represented as sequences of zero or more expressions; the receiver is treated as the first method argument. The `.state` and `.retval` expressions denote the state of the receiver after the invocation and the result of the invocation, respectively. For brevity, Fig-

ure 2 does not specify types, but the expressions are well-typed according to the Java typing rules [1].

For example, `s2` at the end of T2 is represented as `push(push(<init>().state, 3).state, 5).state`, where `<init>` represents the constructor that takes no receiver and `<init>().state` represents the object created by the constructor invocation. This object becomes the receiver of the method invocation `push(3)`, and so on.

Some of our techniques represent method-entry states using tuples of expressions. Two tuples are equivalent iff their expressions are component-wise identical.

Our method-sequence approach allows the tests to contain loops, arithmetic, aliasing, and/or polymorphism. Consider the following manually written tests T4 and T5:

```
Test 4 (T4):                    Test 5 (T5):
  IntStack t = new IntStack();    IntStack s5 = new IntStack();
  IntStack s4 = t;                int i = 0;
  for (int i = 0; i <= 1; i++)    s5.push(i);
    s4.push(i);                   s5.push(i + 1);
```

Our current implementation dynamically monitors the invocations of the methods on the actual objects created in the tests and collects the actual argument values for these invocations. It represents each object using a method sequence; for example, it represents both `t`/`s5` at the end of T4/T5 as `push(push(<init>().state, 0).state, 1).state`.

In future work, we plan to add a static analysis that can gather the method sequence without executing the test code. Although this static analysis would be conservative and less accurate than the dynamic analysis, it would enable detecting some redundant tests without executing them.

3.2.2. Concrete States Each execution of a test operates on the program state that includes a program heap. Our concrete-state approach considers only parts of the heap; 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-labeled object fields.)

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

We define heap isomorphism as graph isomorphism based on node bijection [3].

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

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

Note that the definition allows only nodes to vary: two isomorphic heaps have the same fields for all objects and the same values for all primitive fields.

Some techniques represent state with *rooted* heaps.

```
Map ids; // maps nodes into their unique ids
int[] linearize(Node root, Heap <O,E>) {
  ids = new Map();
  return lin(root, <O,E>);
}
int[] lin(Node root, Heap <O,E>) {
  if (ids.containsKey(root))
    return singletonSequence(ids.get(root));
  int id = ids.size() + 1;
  ids.put(root, id);
  int[] seq = singletonSequence(id);
  Edge[] fields = sortByField({ <root, f, o> in E });
  foreach (<root, f, o> in fields) {
    if (isPrimitive(o))
      seq.add(uniqueRepresentation(o));
    else
      seq.append(lin(o, <O,E>));
  }
  return seq;
}
```

Figure 3. Pseudo-code of linearization

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.

The techniques construct a rooted heap from a program heap $\langle O, E\rangle$ and a tuple $\langle v_0, \ldots, v_n\rangle$ of pointers and primitive values $v_i \in O \cup P$, where $0 \leq i \leq n$. The construction first creates the heap $h' = \langle O', E'\rangle$, where $O' = O \cup \{r\}$ and $E' = E \cup \{\langle r, i, v_i\rangle | 0 \leq i \leq n\}$; $r \notin O$ is the root object. It then creates the rooted heap $\langle r, h\rangle$, where $h = \langle O_h, E_h\rangle$ is the subgraph of h' that contains all nodes reachable from r and their edges, i.e., $O_h \subseteq O'$ is the set of all objects reachable from r within h' and $E_h = \{\langle o, f, o'\rangle \in E' | o \in O_h\}$.

Although there is no polynomial-time algorithm known for checking isomorphism of general graphs, it is possible to efficiently check isomorphism of rooted heaps. Our implementation *linearizes* heaps into sequences such that checking heap isomorphism corresponds to checking sequence equality. Figure 3 shows the pseudo-code of the linearization algorithm. It traverses the entire heap depth first, starting from the root. When it first visits a node, it assigns a unique identifier to the node, keeping this mapping in `ids` to use again for nodes that appear in cycles. Similar linearization was used in model checking [15, 22]. It is easy to show that the linearization normalizes rooted heaps.

Theorem 4 *Two rooted heaps* $\langle o_1, h_1\rangle$ *and* $\langle o_2, h_2\rangle$ *are isomorphic iff* `linearize`$(o_1, h_1) =$`linearize`(o_2, h_2).

3.3. Techniques

Table 1 shows the techniques that we compare. Different techniques use different representations for method-entry states and different comparisons for equivalent states. Each method-entry state describes the receiver object and arguments before a method invocation. We next explain details of all five techniques.

technique	representation	comparison
WholeSeq	the entire method sequence	equality
ModifyingSeq	a part of the method sequence	equality
WholeState	the entire concrete state	isomorphism
MonitorEquals	a part of the concrete state	isomorphism
PairwiseEquals	the entire concrete state	`equals`

Table 1. State representation and comparison

3.3.1. WholeSeq This technique uses the method-sequence approach to represent state. It represents each object with an expression that includes *all* methods invoked on the object since it has been created, including the constructor. Our implementation obtains this representation by executing the tests and keeping a mapping from objects to their corresponding expressions.

Each method-entry state is simply a tuple of expressions that represent the receiver object and the arguments. Two states are equivalent iff the tuples are identical. For example, WholeSeq represents the states before `push(2)` in T3 and T1 as `<push(<init>().state, 3).state, 2>` and `<push(isEmpty(<init>().st).st, 3).st, 2>`, respectively, and these two states are not equivalent.

3.3.2. ModifyingSeq This technique also uses the method-sequence approach. However, it represents each object with an expression that includes *only* those methods that modified the state of the object since it has been created, including the constructor. Our implementation monitors the method executions to determine at run time if some execution modifies the state or not. (Details are in Section 4.)

ModifyingSeq builds and compares method-entry states in the same way as WholeSeq, but since ModifyingSeq uses a coarser representation for objects, it can find more method-entry states to be equivalent. For example, `isEmpty` does not modify the state of the stack, so ModifyingSeq represents states before `push(2)` in both T3 and T1 as `<push(<init>().state, 3).state, 2>` and thus finds them to be equivalent.

3.3.3. WholeState This technique represents method-entry states using the entire concrete state reachable from the receiver object and the arguments. Assume that a test execution is about to invoke `a0.m(a1, ..., an)` and the program heap is $\langle O, E\rangle$. The execution has already evaluated the receiver object and the arguments to some values $v_i \in O \cup P$, where $0 \leq i \leq n$. (Recall that P is the set of all primitive values.) WholeState represents the method-entry state with the rooted heap obtained from $\langle O, E\rangle$ and $\langle v_0, \ldots, v_n\rangle$. Two states are equivalent iff the rooted heaps are isomorphic.

3.3.4. MonitorEquals This technique leverages user-defined `equals` methods to extract only the relevant parts of the state. Like WholeState, MonitorEquals also represents a state with a rooted heap, but this heap is only a subgraph of the entire rooted heap. Conceptually, MonitorEquals first obtains the entire rooted heap from the program heap and the values $\langle v_0, \ldots, v_n\rangle$ of the receiver and arguments (as in WholeState). It then invokes v_i`.equals(`v_i`)` for each non-primitive v_i and monitors the field accesses that these executions make. The rationale behind MonitorEquals is that these executions access only the relevant object fields that define an abstract state. (The executions always return `true` for properly implemented `equals` methods.)

MonitorEquals represents each method-entry state as a rooted heap whose edges consist only of the accessed fields and the edges from the root. Formally, let $\langle r, \langle O, E\rangle\rangle$ be the entire rooted heap and $E_a \subseteq E$ be the set of all fields from E that are accessed during `equals` executions. (The executions may additionally allocate temporary objects and access their fields, but these fields are not in E and these objects are unreachable at the end of the executions.) The method-entry state is the rooted heap $\langle r, \langle O', E'\rangle\rangle$, where $E' = E_a \cup \{\langle o, f, o'\rangle | o = r \wedge \langle o, f, o'\rangle \in E\} \subseteq E$ and $O' = \{o | \langle o, f, o'\rangle \in E' \vee \langle o', f, o\rangle \in E'\} \subseteq O$. In MonitorEquals, two states are equivalent iff their rooted heaps are isomorphic.

For illustration, recall the example and consider the state of stacks before `push(5)` in T1 and T2. The whole concrete state of `s1`/`s2` is shown in the left/right column:

```
// s1 before push(5)        // s2 before push(5)
store = @766a24             store = @11ff43
store.length = 10           store.length = 10
store[0] = 3                store[0] = 3
store[1] = 2                store[1] = 0
store[2] = 0                store[2] = 0
...                         ...
store[9] = 0                store[9] = 0
size = 1                    size = 1
```

where the values of the `store` array are their identifiers (reference addresses, prefixed with `@`). These states are not equivalent, because `store[1]` differs. However, the execution of `this.equals(this)` accesses only the fields `size`, `store`, and elements of `store` whose indices are up to the value of `size`. In this example, the accessed part of `s1`/`s2` is shown in the left/right column:

```
// this.equals(this)        // this.equals(this)
// before s1.push(5)        // before s2.push(5)
store = @766a24             store = @11ff43
store[0] = 3                store[0] = 3
size = 1                    size = 1
```

These two states are not identical, as the addresses differ, but they are isomorphic. Thus, MonitorEquals reports that the method-entry states before `push(5)` in T1 and T2 are equivalent.

3.3.5. PairwiseEquals This technique also leverages user-defined `equals` methods to detect equivalent states. It implicitly uses the entire program heap to represent method-entry states. However, it does not compare (parts of) states by isomorphism. Instead, it runs the test to build the concrete objects that correspond to the receiver and arguments, and then uses the `equals` method to compare pairs of states. Two states s_1 and s_2 are equivalent iff s_1`.equals(`s_2`)` returns `true`.

This technique can find more equivalent objects than the previous technique. For example, consider a class that implements a set using an array. PairwiseEquals reports two objects to be equivalent if they have the same set of array elements, regardless of the order, whereas MonitorEquals reports two objects with the same elements but different order to be non-equivalent. However, PairwiseEquals is typically slower as it compares the whole state, whereas MonitorEquals compares only parts of the state, and additionally Rostra uses efficient hashing and storing in MonitorEquals, because we know the representation (sequence).

3.4. Redundant Tests

Each test execution produces several method executions.

Definition 5 *A* method execution $\langle m, s\rangle$ *is a pair of a method* m *and a method-entry state* s*.*

We denote with $[\![t]\!]$ the sequence of method executions produced by a test t, and we write $\langle m, s\rangle \in [\![t]\!]$ when a method execution $\langle m, s\rangle$ is in the sequence for t. We define equivalent method executions based on equivalent states.

Definition 6 *Two method executions* $\langle m_1, s_1\rangle$ *and* $\langle m_2, s_2\rangle$ *are* equivalent *iff* $m_1 = m_2$ *and* s_1 *and* s_2 *are equivalent.*

We further consider minimal test suites that contain no redundant tests.

Definition 7 *A test* t *is* redundant *for a test suite* S *iff for each method execution of* $[\![t]\!]$*, there exists an equivalent method execution of some test from* S*.*

Definition 8 *A test suite* S *is* minimal *iff there is no* $t \in S$ *that is redundant for* $S \backslash \{t\}$*.*

Minimization of a test suite S' finds a minimal test suite $S \subseteq S'$ that exercises the same set of non-equivalent method executions as S' does.

Definition 9 *A test suite* S minimizes *a test suite* S' *iff* S *is minimal and for each* $t' \in S'$ *and each* $\langle m', s'\rangle \in [\![t']\!]$*, there exist* $t \in S$ *and* $\langle m, s\rangle \in [\![t]\!]$ *such that* $\langle m', s'\rangle$ *and* $\langle m, s\rangle$ *are equivalent.*

Given a test suite S', there can be several test suites $S \subseteq S'$ that minimize S'. Our implementation uses a greedy algorithm to find one of the test suites that minimizes S'.

4. Implementation

We have implemented the five Rostra techniques for collecting method-entry states and comparing equivalence in Java. Our current implementation collects method-entry states dynamically during test executions. We use the Byte Code Engineering Library [8] to instrument the bytecodes of the classes under test at the class-loading time. The instrumentation adds the code for collecting state representations at the entry of each method call in a test. It also adds the code for monitoring instance-field reads and writes.

Our instrumentation collects the method signature, the receiver-object reference, and the arguments at the beginning of each method call in the test. The receiver of these calls is usually an instance object of the class under test. The instrumentation does not collect the method-entry states for calls that are internal to these objects. Different techniques also collect and maintain additional information.

The WholeSeq and ModifyingSeq techniques maintain a table that maps each object to a method sequence that represents that object. At the end of each method call, the sequence that represents the receiver object is extended with the appropriate information that represents the call, unless the method execution has not modified the receiver, in which case ModifyingSeq does not extend the sequence. ModifyingSeq dynamically monitors the execution and determines that the receiver is modified if there is a write to a field that is reachable from the receiver.

The WholeState technique uses Java reflection [1] to recursively collect all the fields that are reachable from the receiver and arguments before the method call. The MonitorEquals technique executes v_i`.equals(`v_i`)` for the receiver and each non-primitive argument v_i before the method call. It then monitors these executions to collect all fields that are accessed. (The MonitorEquals technique needs to carefully avoid the common optimization pattern that compares the receiver and the argument for identity `this == that` within `equals` methods.) To compare states, WholeState and MonitorEquals use our implementation of the linearization algorithm (Section 3.2.2). The PairwiseEquals technique creates the objects for the receiver and arguments and then compares them using `equals` methods. Note that subsequent test execution can modify these objects, so PairwiseEquals needs to reproduce them for comparison. Our current implementation re-executes method sequences; an alternative would be to maintain a copy of the objects.

5. Applications

We propose these four applications of Rostra: test-suite assessment, test selection, test-suite minimization, and test generation.

Assessment: Rostra provides a novel quantitative comparison of test suites, especially those generated by automatic test-generation tools. For each test suite, Rostra can find non-equivalent object states, non-equivalent method executions, and non-redundant tests. We can then use their metrics to compare the quality of different test suites.

Selection: We can use Rostra to select a subset of automatically generated tests to augment an existing (manually or automatically generated) test suite. We feed the existing test suite and the new tests to Rostra, running the existing test suite first. The minimal test suite that Rostra then produces will contain those new tests that are non-redundant with respect to the existing test suite.

Minimization: We can use Rostra to minimize an automatically generated test suite for correctness inspection and regression executions. Without a priori specifications, automatically generated tests typically do not have test oracles for correctness checking, and the tester needs to manually inspect the correctness of (some) tests. Rostra helps the tester to focus only on the non-redundant tests, or more precisely the non-equivalent method executions. Running redundant tests is inefficient, and Rostra can remove these tests from a regression test suite. However, we need to be careful because changing the code can make a test that is redundant in the old code to be non-redundant in the new code. If two method sequences in the old code produce equivalent object states, *and* the code changes do not impact these two method sequences [25], we can still safely determine that the two sequences in the new code produce equivalent object states. Additionally, we can always safely use Rostra to perform regression test prioritization [24] instead of test-suite minimization.

Generation: Existing test-generation tools can incorporate Rostra to avoid generating and executing redundant tests. Although our five Rostra techniques are dynamic, they can determine whether a method execution me is equivalent to some other execution *before* running me; the method-entry state required for determining equivalence is available before the execution. Test-generation tools that execute tests, such as Jtest [21], can easily integrate Rostra. Jtest executes already generated tests and observes their behavior to guide the generation of future tests. Running Jtest is currently expensive—it spends over 10 minutes generating the tests for relatively large classes in our experiments (Section 6)—but much of this time is spent on redundant tests.

We have implemented a prototype test-generation tool based on Rostra [28]. It dynamically and iteratively generates non-redundant tests to exercise non-equivalent object states. Our prototype performs combinatorial testing by generating tests that exercise each possible combination of non-equivalent method, receiver, and arguments. Our preliminary results show that our prototype generates test suites better by several metrics than Jtest.

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

Table 2. Experimental subjects

6. Experiments

This section presents two experiments that assess how well Rostra detects redundant tests: 1) we investigate the benefit of applying Rostra on tests generated by existing tools; and 2) we validate that removing redundant tests identified by Rostra does not decrease the quality of test suites. We have performed the experiments on a Linux machine with a Pentium IV 2.8 GHz processor using Sun's Java 2 SDK 1.4.2 JVM with 512 MB allocated memory.

6.1. Experimental Setup

Table 2 lists the 11 Java classes that we use in our experiments. The `IntStack` class is our running example. The `UBStack` class is taken from the experimental subjects used by Stotts et al. [26]. The `ShoppingCart` class is a popular example for JUnit [6]. The `BankAccount` class is an example distributed with Jtest [21]. The remaining seven classes are data structures used to evaluate Korat [3, 19]. 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.

We use two third-party test generation tools, Jtest [21] and JCrasher [7], to automatically generate test inputs for program subjects. 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 last two columns of Table 2 show the number of tests generated by Jtest and JCrasher.

Our first experiment uses the five Rostra techniques to detect redundant tests among those gener-

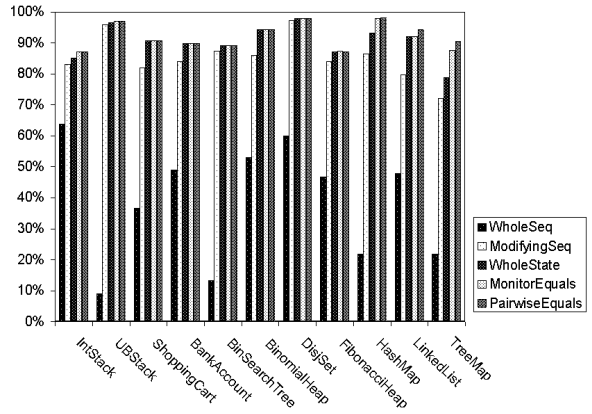

Figure 4. Percentage of redundant tests among Jtest-generated tests

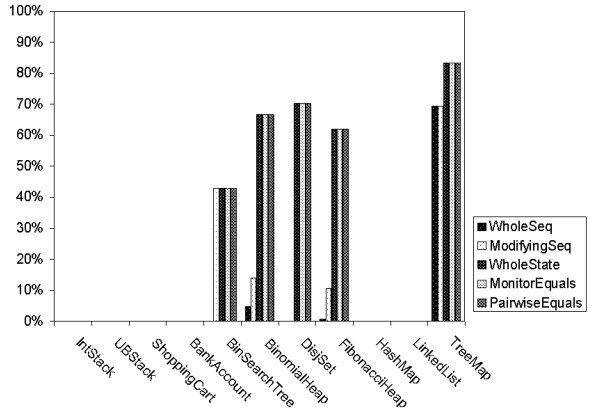

Figure 5. Percentage of redundant tests among JCrasher-generated tests

ated by Jtest and JCrasher. Our second experiment compares the quality of original and Rostra-minimized test suites using 1) branch coverage, 2) non-equivalent, uncaught-exception count, and 3) fault-detection capability. We adapt Hansel [12] to measure branch coverage and non-equivalent, uncaught-exception count. (Two exceptions are equivalent if they have the same throwing location and type.) To estimate the fault-detection capability, we use two mutation-analysis tools for Java: Jmutation [18] and Ferastrau [19]. We select the first 300 mutants produced by Jmutation and configure Ferastrau to produce around 300 mutants for each subject. We have written specifications and used the JML runtime verifier [5] to compare the method-exit states and returns of the original and mutated method executions.

6.2. Experimental Results

Figures 4 and 5 show the results of the first experiment—the percentage of redundant tests generated—for Jtest and JCrasher, respectively. We observe that all techniques except WholeSeq identify around 90% of Jtest-generated tests to be redundant for all subjects and 50% of JCrasher-generated tests to be redundant for five out of 11 subjects. Possible reasons for higher redundancy of Jtest-generated tests include: 1) Jtest generates more tests; and 2) Jtest-generated tests have longer call length.

The two method-sequence techniques identify fewer redundant tests than the three concrete-state techniques, and there is no significant difference in the results for the latter three techniques. We hypothesize that our experimental subjects do not have many irrelevant object fields for defining object states and/or the irrelevant object fields do not significantly affect the redundant test detection.

We also measured the percentages of equivalent object states and equivalent method executions; they have similar distributions as the redundant tests.

The elapsed real time of running our implementation is reasonable: it ranges from a couple of seconds up to several minutes, determined primarily by the class complexity and the number of generated tests. To put this time into perspective, we need to consider the whole test generation: if test-generation tools such as Jtest incorporated Rostra into generation, the time savings achieved by avoiding redundant tests would significantly exceed the extra cost of running Rostra [28].

Table 3 shows the results of the second experiment: non-equivalent, uncaught-exception counts (columns 2 and 3), branch-coverage percentages (columns 4 and 5), killing rates for Ferastrau mutants (columns 6 and 7), and killing rates for Jmutation mutants (columns 8 and 9). The columns marked “jte” and “jcr” correspond to Jtest and JCrasher, respectively. The original Jtest-generated and JCrasher-generated test suites have the same measures as their corresponding Rostra-minimized test suites in all cases except for the four cases whose entries are marked with “*”. The differences are due only to the MonitorEquals and PairwiseEquals techniques. The minimized Jtest-generated test suites for `IntStack` and `TreeMap` cannot kill three Ferastrau-generated mutants that the original test suites can kill. This shows that minimization based on `equals` can reduce the fault-detection capability of a test suite, but the probability is very low. The minimized Jtest-generated test suites for `HashMap` and `TreeMap` cannot cover two branches that the original test suites can cover. We have reviewed the code and found that two fields of these classes are used for caching; these fields do not affect object equivalence (defined by `equals`) but do affect branch coverage. These four cases suggest a further in-

class	excptn count		branch cov [%]		Ferastrau kill [%]		Jmutation kill [%]	
	jte	jcr	jte	jcr	jte	jcr	jte	jcr
IntStack	1	1	67	50	*45	40	24	23
UBStack	2	0	94	56	57	25	78	37
ShoppingCart	2	1	93	71	57	51	80	20
BankAccount	3	3	100	100	98	98	89	89
BinSearchTree	3	0	67	14	33	5	57	11
BinomialHeap	3	3	90	66	89	34	64	48
DisjSet	0	0	61	51	26	18	40	29
FibonacciHeap	2	2	86	58	73	21	68	35
HashMap	1	1	*72	43	52	23	48	24
LinkedList	19	10	79	48	24	7	25	9
TreeMap	4	3	*33	11	*16	4	16	7

Table 3. Quality of Jtest-generated, JCrasher-generated, and minimized test suites

vestigation on the use of `equals` methods in detecting redundant tests as future work.

6.3. Threats to Validity

The threats to external validity primarily include the degree to which the subject programs and third-party test generation tools are representative of true practice. Our subjects are from various sources and the Korat data structures have nontrivial size for unit tests. Of the two third-party tools, one—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 implementation, Jtest, JCrasher, or other measurement tools might cause such effects. To reduce these threats, we have manually inspected the collected execution traces for several program subjects.

7. Related Work

Rostra techniques are related to work on state representation and comparison, and Rostra itself is related to work on test selection and minimization. (Test generation based on Rostra is also related to work on test generation [28].)

Iosif [15] and Robby et al. [22] use linearization to encode states in model checkers. They do not apply any technique as our MonitorEquals to collect only the relevant object fields, but always collect all fields. Zimmermann and Zeller use a memory graph and its visualization to represent and explore states during C program executions [30]. They reduce the comparison of program states to the comparison of graphs.

Most of the previous work on detecting equivalence of object states [2, 10, 14] has developed techniques based on observational equivalence [17]. These techniques are typically used to infer axioms in algebraic specifications or verify their correctness, while our Rostra techniques are used to detect redundant tests. Previous techniques are typically much slower than our techniques, but our non-`equals`-based techniques can find fewer equivalent objects (too conservative) and our `equals`-based techniques can, depending on `equals`, find more equivalent objects (unsound) than observational equivalence.

Grieskamp et al. present the AsmLT test-generation tool that incorporates test selection [11]. AsmLT allows the user to provide an abstraction function (α) that maps states of abstract state machines into so-called "hyperstates"; two tests are equivalent if they lead to the states (s_1 and s_2) that map to the same hyperstate ($s_1 \equiv s_2 \Leftrightarrow \alpha(s_1) = \alpha(s_2)$). Rostra, instead, allows the user to define equivalence more directly via a binary, boolean-returning method (m) that takes two states (s_1 and s_2) and determines their equivalence ($s_1 \equiv s_2 \Leftrightarrow m(s_1, s_2)$). In practice, the existing `equals` methods suffice and Rostra uses them fully automatically, but in principle, Rostra allows the user to provide other methods for equivalence. Moreover, our WholeState and MonitorEquals techniques compare states using isomorphism, whereas AsmLT always uses equality. Our test-generation tool based on Rostra [28] uses combinatorial generation similar to AsmLT, with different selections.

Chang and Richardson use specification-coverage criteria for selecting tests that exercise new aspects of a priori provided unit specifications [4]. Without requiring a priori specifications, Harder et al. use the operational difference technique to augment and minimize regression test suites [13]. Our previous work uses operational violations to select a small valuable subset of automatically generated tests for manual inspection [29]. Clustering and sampling the execution profiles are also used to select tests for inspection [9]. There are also several approaches to minimizing [23] or prioritizing [24] tests for regression testing, primarily based on structural coverage. Rostra complements these existing approaches based on specifications or structural coverage. These approaches typically select fewer tests than Rostra, but Rostra differs in that it aims to select tests that preserve the quality of the original test suite.

8. Conclusion

We have proposed Rostra, a novel framework for detecting redundant object-oriented unit tests, and presented five techniques within this framework. We have proposed four practical applications of the framework. We have conducted experiments that evaluate the effectiveness of Rostra on detecting redundant tests in test suites generated by two third-party test-generation tools. The results show that Rostra can substantially reduce the size of these test suites with-

out decreasing their quality. These results strongly suggest that tools and techniques for generation of object-oriented test suites must consider avoiding redundant tests.

Acknowledgments

We thank Yu-Seung Ma and Jeff Offutt for providing the Jmutation tool. We thank Andrew Petersen, Vibha Sazawal, and the anonymous reviewers for their valuable feedback on an earlier version of this paper. This work was supported in part by the National Science Foundation under grants ITR 0086003 and CCR00-86154. We acknowledge support through the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298.

References

[1] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[2] G. Bernot, M. C. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *Softw. Eng. J.*, 6(6):387–405, 1991.

[3] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 123–133, 2002.

[4] J. Chang and D. J. Richardson. Structural specification-based testing: Automated support and experimental evaluation. In *Proceedings of the 7th ESEC/FSE*, pages 285–302, 1999.

[5] Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The JML and junit way. In *Proc. European Conference on Object-Oriented Programming (ECOOP)*, pages 231–255, 2002.

[6] M. Clark. Junit primer. Draft manuscript, October 2000.

[7] C. Csallner and Y. Smaragdakis. Jcrasher documents. Online manual, December 2003.

[8] M. Dahm and J. van Zyl. Byte code engineering library, April 2003.

[9] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 339–348, 2001.

[10] 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.

[11] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating finite state machines from abstract state machines. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 112–122, 2002.

[12] Hansel 1.0, 2003. `http://hansel.sourceforge.net/`.

[13] M. Harder, J. Mellen, and M. D. Ernst. Improving test suites via operational abstraction. In *Proceedings of the 25th International Conference on Software Engineering*, pages 60–71, 2003.

[14] J. Henkel and A. Diwan. Discovering algebraic specifications from Java classes. In *17th European Conference on Object-Oriented Programming*, pages 431–456, 2003.

[15] R. Iosif. Symmetry reduction criteria for software model checking. In *Proceedings of the 9th SPIN Workshop on Software Model Checking*, pages 22–41, July 2002.

[16] 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.

[17] B. Liskov and J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2000.

[18] Y.-S. Ma, Y.-R. Kwon, and J. Offutt. Inter-class mutation operators for Java. In *Proceedings of International Symposium on Software Reliability Engineering*, pages 352–363, 2002.

[19] 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.

[20] B. Meyer. *Eiffel: The Language*. Prentice Hall, New York, N.Y., 1992.

[21] Parasoft. Jtest manuals version 4.5. Online manual, October 2002.

[22] Robby, M. Dwyer, J. Hatcliff, and R. Iosif. Space-reduction strategies for model checking dynamic software. In *Proceedings of the 2003 Workshop on Software Model Checking*, volume 89 of *ENTCS*, 2003.

[23] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Proc. the International Conference on Software Maintenance*, pages 34–43, 1998.

[24] G. Rothermel, R. J. Untch, and C. Chu. Prioritizing test cases for regression testing. *IEEE Trans. Softw. Eng.*, 27(10):929–948, 2001.

[25] B. G. Ryder and F. Tip. Change impact analysis for object-oriented programs. In *Proceedings of ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 46–53, 2001.

[26] D. Stotts, M. Lindsey, and A. Antley. An informal formal method for systematic junit test case generation. In *Proceedings of the 2002 XP/Agile Universe*, pages 131–143, 2002.

[27] Sun Microsystems. *Java 2 Platform, Standard Edition, v1.3.1 API Specification*. `http://java.sun.com/j2se/1.3/docs/api/`.

[28] T. Xie, D. Marinov, and D. Notkin. Improving generation of object-oriented test suites by avoiding redundant tests. Technical Report UW-CSE-04-01-05, University of Washington Department of Computer Science and Engineering, Seattle, WA, January 2004.

[29] T. Xie and D. Notkin. Tool-assisted unit test selection based on operational violations. In *Proceedings of 18th IEEE International Conference on Automated Software Engineering*, pages 40–48, 2003.

[30] T. Zimmermann and A. Zeller. Visualizing memory graphs. In *the Dagstuhl Seminar on Software Visualization*, volume 2269 of *LNCS*, pages 191–204, 2001.