Improving Generation of Object-Oriented Test Suites by Avoiding Redundant Tests

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

[1] Department of Computer Science and Engineering, University of Washington, Seattle, WA 98195, USA
{taoxie,notkin}@cs.washington.edu
[2] MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA 02139, USA
marinov@lcs.mit.edu

Technical Report UW-CSE-04-01-05

January 2004

ABSTRACT

Object-oriented 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 object states. These tools generate redundant tests that exercise the same method behavior, which increases the testing time without increasing the ability to detect faults.

We propose a formal framework for detecting redundant tests and present five fully automatic techniques within this framework. Based on these techniques, we have developed a test-minimization tool that removes redundant tests from test suites and a test-generation tool that iteratively augments test suites with non-redundant tests. We evaluate our tools on eight subjects taken from a variety of sources. The experimental results show that our test minimization can remove over 90% of the tests generated by Jtest for most subjects and 30% of the tests generated by JCrasher for half of the subjects, without decreasing the quality of test suites. The results also show that our test generation can effectively generate new tests that increase the quality of test suites generated by Jtest and JCrasher.

1. INTRODUCTION

Most existing tools for automatic generation of object-oriented test suites, such as Jtest [26] (a commercial tool for Java) and JCrasher [9] (a research prototype for Java), test a class by generating sequences of method invocations for the class. 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 same methods on such equivalent objects is redundant. Since testing is typically constrained by time limits, a key issue is to avoid redundant tests that only increase the testing time and do not increase the ability to detect faults.

We propose a formal framework for detecting redundant tests based on equivalent objects. Within this framework, we present five fully automatic techniques for detecting equivalent objects. These techniques do not require any user input, except that two of the techniques assume that the class under test implements an equality method. In Java, the `equals` method is defined in the `java.lang.Object` class and often overridden in subclasses; it is used pervasively, for example to compare elements in the Java Collections Framework [31]. Any `equals` method must satisfy a set of properties, such as implementing an equivalence relation; otherwise, the collections do not behave as expected.

Some testing tools, such as AsmLT [13, 15], require the user to provide an abstraction function [24] for classes; two objects are then equivalent if they map to the same abstract value. We can view our techniques as automatically defining an abstraction function based on `equals`; it is more conservative than a user-provided abstraction function, but fully automatic. Several other projects [4, 11, 17] use observational equivalence [24] to define equivalent objects. However, checking observational equivalence is very expensive: by definition it takes infinite time, but in practice approximations are used. Our techniques sometimes give more conservative results than observational equivalence, but they take much less time.

We have developed a test-minimization tool that removes redundant tests from a test suite and a test-generation tool that iteratively augments test suites with non-redundant tests. During the execution of an existing test suite, our test-generation tool monitors and collects method sequences that lead to each non-equivalent object state. It also monitors and collects method arguments exercised by the existing test suite. The tool then uses a form of combinatorial testing that can exhaustively exercise each non-equivalent object state for all methods and all collected arguments. Our test generation exploits the knowledge of existing tests to generate arguments; this is a novel approach that complements generation of arguments based on default values [9, 26] or user-defined values [5, 13, 26]. During this combinatorial generation, new non-equivalent objects are encountered, and the process iteratively continues until for a user-defined number of iterations. The test generation guarantees that each generated test is non-redundant.

This paper makes the following main contributions:

- We propose a formal framework for detecting equivalent object states and redundant tests.
- We present five techniques within this framework.
- We develop a test-minimization tool that removes redundant tests from a test suite without sacrificing the quality of the test suite.
- We develop a test-generation tool that iteratively augments test suites with non-redundant tests. The tool does not require users to define any extra input for new tests, but fully exploits the information from the existing test suite.

- We evaluate our test minimization and generation on eight subjects taken from a variety of sources. The experimental results show that our test minimization can remove over 90% of the tests generated by Jtest for most subjects and 30% of the tests generated by JCrasher for half of the subjects. The results also show that our test generation can effectively add tests that increase branch coverage 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 shown in Figure 1 and taken from Henkel and Diwan [17]. The array `store` contains the elements of the stack, and `size` is the number of the elements and the index of the first free location in the stack. The method `push`/`pop` appropriately increases/decreases the size after/before writing/reading the element. Additionally, `push`/`pop` grows/shrinks the array when the `size` is equal to the whole/half of the array length. The method `isEmpty` is an observer that checks if the stack has any elements, and the method `equals` compares two stacks for equality.

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 several method sequences 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 [26] and JCrasher [9]. For such tests, the correctness checking depends on design-by-contract annotations [23, 25]. If the code has annotations, the tools translate them into run-time assertions [7, 26] that are evaluated during the execution. If there are no annotations, the tools only check the robustness of the code: they execute the tests to check if any uncaught exception is thrown [9].

To determine redundant tests, our techniques monitor executions of the tests. An execution of a test consists of transitions on the state of the Java program. Our techniques track these transitions at the granularity of methods: each test-case 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 state and several approaches for determining *equivalent* states (Section 3).

```
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;
    this.size++;
  }
  public int pop() {
    int result = this.store[this.size - 1];
    this.size--;
    if (this.store.length > INITIAL_CAPACITY
     && this.size * 2 < this.store.length) {
      int[] store = new int[this.store.length / 2];
      System.arraycopy(this.store, 0, store, 0, this.size);
      this.store = store;
    }
    return result;
  }
  public boolean isEmpty() {
    return (this.size == 0);
  }
  public boolean equals(Object other) {
    if (other == null) return false;
    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 in Java

We call two method executions equivalent if they are invocations of the same method with respective equivalent states. Our test-minimization and test-generation approaches are concerned with *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 4 presents our test-minimization approach that removes redundant tests from a test suite and our test-generation approach that generates only non-redundant tests.

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. These tools typically consider two tests to be equivalent only if they are identical. The technique represents state using method sequences that create objects and compares state using sequence equality. It finds all three example tests to be non-redundant.
- ModifyingSeq: This technique improves on the previous by using in state representation only those method invocations that 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 (defined in Section 3). 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 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 next show how equivalent states give rise to equivalent method executions and define redundant tests and test-suite minimization. We finally discuss the assumptions that our techniques make about the code under test.

3.1 State Representation and Comparison

Our techniques use two main approaches for state representation: 1) method sequences and 2) concrete state of the objects. Both approaches assume that the classes under test have some set of methods (distinguished not only by their Java name, but by the entire signature) and consider constructors as methods.

3.1.1 Method Sequences

Each execution of a test creates several objects and invokes methods on these objects. Our method-sequence approach represents state using sequences of method invocations to represent objects, following Henkel and Diwan who use the representation in mapping Java classes to algebras [17]. The state representation uses symbolic expressions whose concrete grammar is shown in Figure 2. Each object and value are represented with an expression. Arguments for method invocations are represented as sequences of zero of more expressions; the receiver is treated as the first method argument. This representation assumes that every method invocation can only modify the state of the receiver object (and not any of the arguments) and return a result [17]. The `.state` and `.retval` expressions denote the state of the receiver after the method invocation and the result of the invocation, respectively. For brevity, Figure 2 does not specify types, but the expressions are well-typed according to the Java typing rules [1].

For example, the object `s2` at the end of T2 is represented as[1] `push(push(<init>().state, 3).state, 5).state`, where `<init>` represents the constructor method. A constructor takes no receiver, and `<init>().state` represents the object state created by the constructor-method invocation. This object state 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.

Some tests may contain loops or arithmetic operations. For example, two manually written tests T4 and T5 are shown in the left and right columns respectively:

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

[1]More precisely, each method is represented together with its signature such that different methods have different representation, even if they have the same name.

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

Figure 2: Grammar for symbolic expressions

Our method-sequence approach monitors the invocations of the methods of `s4`/`s5` from T4/T5, and collects these method calls' actual argument values. The approach represents the object states at the end of both T4 and T5 as

`push(push(<init>().state, 0).state, 1).state` .

3.1.2 Concrete State

Each execution of a test operates on the program state that includes a program heap. Our concrete-state approach considers only parts of the program heap. We also call each part a "heap" and view it as a graph: nodes represent objects and edges represent object fields. Let P be the set consisting of all primitive values, including `null`, integers, booleans etc. Let O be a set of objects whose fields form a set F. (Array elements are considered as object fields labelled with indices.)

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.

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:*

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

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 of our techniques use *rooted* heaps as state representation.

DEFINITION 3. *A rooted heap is a pair* $\langle r, h\rangle$ *of a root object* r *and a heap* h *such that all nodes in* h *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 $0 \le i \le n$. The construction first creates the heap $h' = \langle O', E'\rangle$, where $O' = O \cup \{r\}$ and $E' = E \cup \{\langle r, i, v_i\rangle | 0 \le i \le n\}$; $r \notin O$ 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, edge-labelled graphs. Our approach is to *linearize* 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 in the depth-first order, starting from the root. When it first visits a node, it assigns a unique identifier to the node, keeping this mapping in `ids`. If there is a cycle in the heap, the traversal visits some nodes several time and uses previously assigned identifiers to represent nodes. Similar linearization has been applied in model checking for encoding states [19, 27, 33].

It is easy to show that the linearization normalizes rooted heaps into sequences.

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).

3.2 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.

3.2.1 *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 maintaining 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, this technique represents the states before `push(2)` in T3 and T1 as

```
        <push(<init>().state, 3).state, 2>
```

and

```
<push(isEmpty(<init>().state).state, 3).state, 2>,
```

respectively. According to this technique, these two states are not equivalent.

3.2.2 *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 executes the tests and determines at run time if a method modifies the state or not.

This technique builds and compares method-entry states the same as the previous technique. However, since it uses a coarser representation for objects, it can find more method-entry states to be equivalent. For example, since `isEmpty` does not modify the state of the stack, this technique represents states before `push(2)` in both T3 and T1 as

```
        <push(<init>().state, 3).state, 2>
```

and thus finds them to be equivalent.

3.2.3 *WholeState*

This technique represents method-entry states using the entire concrete state reachable from the receiver object and the arguments. Assume that a test-case execution is about to invoke some method `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.) This technique 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.

3.2.4 *MonitorEquals*

This technique leverages user-defined `equals` methods to extract only the relevant parts of the state. Like the previous technique, this technique also represents state with a rooted heap, but this heap is only a subgraph of the entire rooted heap. Conceptually, this technique 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 the previous technique. It then invokes[2] v_i`.equals(`v_i`)` for

[2]This execution always returns `true` for properly implemented `equals` methods.

```
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 the linearization algorithm

each non-primitive v_i and monitors the field accesses that these executions make. The rationale behind this technique is that these invocations of `equals` access the relevant object fields that define an abstract state.

This technique 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[3]. The method-entry state is the rooted heap $\langle r, \langle O', E'\rangle\rangle$, where $E' = E_a \cup \{\langle r, f, o\rangle | \langle r, f, o\rangle \in E\} \subseteq E$ and $O' = \{o | \langle o, f, o'\rangle \in E' \vee \langle o', f, o\rangle \in E'\} \subseteq O$. In this technique, two states are equivalent iff their rooted heaps are isomorphic.

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

```
// s2 before push(5)        // s1 before push(5)
store = @766a24             store = @11ff436
store.length = 10           store.length = 10
store[0] = 3                store[0] = 3
store[1] = 0                store[1] = 2
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 `s2`/`s1` is shown in the left/right column:

```
// this.equals(this)        // this.equals(this)
// before s2.push(5)        // before s1.push(5)
store = @766a24             store = @11ff436
store[0] = 3                store[0] = 3
size = 1                    size = 1
```

These two states are not identical, as the address differs, but they are isomorphic, and thus this technique reports that the method-entry states before `push(5)` in T2 and T1 are equivalent.

3.2.5 *PairwiseEquals*

This technique also leverages user-defined `equals` methods to detect equivalent states. It implicitly uses the entire program heap

[3]The executions may additionally allocate temporary objects and access their fields, but the fields of these objects are not in E and these objects are unreachable at the end of the executions.

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`.

3.3 Redundant Tests

Each execution of a test produces a sequence of 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 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.*

Our test minimization and generation approaches are concerned with redundant tests.

DEFINITION 7. *A test* t *is* redundant *for a test suite* S *iff for each method execution of* t*, exists an equivalent method execution of some test in* S.

A test suite is minimal if it has no redundant test.

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

Given a test suite S, there can be several minimal test suites $S' \subseteq S$. Our minimization uses a greedy algorithm to find one of those minimal test suites S'. We could additionally find a test suite S' that is optimal in that it minimizes the number of tests in S' or the total number of method executions for the tests in S'. These optimization problems, called "minimum set cover" and "minimum exact cover" respectively, are known to be NP complete, and in practice approximation algorithms [20] are used.

3.4 Assumptions

Our techniques make the following assumptions about the code under test:

- Methods are deterministic. (Otherwise, different executions for the same input may produce different results, so model-checking techniques are more applicable than testing.)
- The execution of the methods depends only on the state reachable from the receiver and other arguments. (This means that the code under test does not for example reads files or accesses network.)
- Method-sequence representation additionally assumes that each method can only modify the state of the receiver and return a result.
- Techniques based on the `equals` methods additionally assume that these methods are properly implemented (as per the contract in `java.lang.Object` [31]) and that they are pure, i.e., do not modify any state that they do not temporarily allocate during the execution.

4. IMPLEMENTATION

This section presents details of our implementation. We present the implementation of our techniques for collecting method-entry states and comparing their equivalence. We then present our test-minimization and test-generation tools.

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: Techniques for state representation and comparison

4.1 Techniques

We use the Byte Code Engineering Library (BCEL) [10] to rewrite the bytecodes of a class at class loading time. We collect state representation at the entry and exit of each method call between a candidate object (usually being an instance of the class under test) and its clients. We do not collect object states for those method calls that are internal to the candidate object.

During object state collection, we collect receiver object references, method signatures, and arguments at method entries or returns at method exits of a candidate object. We also instrument test classes to collect receiver object references, method signatures, arguments and return at call sites of those method sequences that lead to argument object states for the candidate object's method. Then we can use the collected method call information to construct the method sequence that leads to a particular state of a candidate object or argument object. The WholeSeq and ModifyingSeq techniques use these constructed method sequences to represent object states. In the implementation of the PairwiseEquals technique, we execute collected method sequences to reproduce object states.

The WholeState technique uses Java reflection mechanisms [1] to recursively collect all the fields that are reachable from a candidate object. The MonitorEquals and ModifyingSeq techniques need to collect accesses of the fields that are reachable from a candidate object. The MonitorEquals technique represents the object state of a non-primitive v_i by using those collected field accesses within v_i`.equals(`v_i`)`. The ModifyingSeq technique determines if a method modifies the receiver object state by observing if there is a write of a field that is reachable from the receiver object. To collect field accesses, we insert some code before each instance field read or write site in the class bytecodes at loading time. The inserted code invokes our runtime analysis routines to collect only those accessed fields that are reachable from a candidate object. Since our field access monitoring and collection are based on the instrumentation of bytecodes at class loading time, if specifications or aspects are weaved into bytecodes by using some tools, such as JML tool-set [7] or AspectJ compiler [32], we can monitor those field accesses in specifications or aspects, and collect their field values as relevant ones for state representation.

In our implementations, all techniques except for the ModifyingSeq technique use the same technique for argument object state representation as the one for receiver object state representation. Due to engineering considerations, we do not collect the field accesses of every receiver object in the method sequence that leads to an argument object for a candidate object's method. Therefore we cannot determine whether a method in this sequence modifies its receiver object state. In our implementations, the technique used to represent argument object states can be different from the one used to represent receiver object states. Therefore we can use any of the other four techniques for representing argument object states when we use the ModifyingSeq technique to represent receiver object states. Indeed, if the intersection of the field reference sets reachable from a receiver object and an argument object is not empty, the accuracy of the concrete-state representation would be compromised when using method-sequence state representation for argument object states.

4.2 Test Minimization

In terms of testing the behavior of a method, equivalent method executions are redundant. However, sometimes redundancy of equivalent method executions is unavoidable in a test suite. For example, in order to test all public method calls of a class on a particular object state, we have to duplicate the method call sequence that produces such an object state several times as many as the number of public state-modifying methods. However, ideally we still expect each test exercises at least one new method execution. If a test in a test suite does not produce at least one new method execution that is not equivalent to any of those produced by previously executed tests in the test suite, we consider this test to be redundant and it is removed from the test suite.

We instrument the entry and exit of each test method in a given test class. In the test method entry, we insert a method call to our runtime analysis routine to notify the beginning of the test. After this test method is executed, our tool collects method executions exercised within this test method, and sees whether the representations of these method executions exist in a trie data structure [14]. The trie is initially empty before running tests in the given test class. If the tool cannot find the collected representation of a method execution in the trie, the tool adds this representation into the trie. In the test method exit, we insert a method call to our runtime analysis routine to notify the end of the test. If within the execution of this test, there exists at least one method execution that has been added to the trie, the test is determined to be a non-redundant one, otherwise, a redundant one. After we execute all the test methods in the test class, we process the source code of the test class by commenting out the source code of redundant test methods, and save the processed source code to a new minimized test class.

4.3 Test Generation

We divide the test generation problem into two sub-problems: object state setup and method parameter generation. Object state setup puts an object of the class under test into a particular state before invoking methods on it. Method parameter generation produces particular arguments for a method to be invoked on the object state.

A *method argument list* is characterized by the method signature and the arguments for the method. Two argument lists are non-equivalent iff their method signatures are different or some of their corresponding arguments are non-equivalent. Unlike a non-equivalent method execution, a non-equivalent method argument list does not include the method-entry state. In method parameter generation, we generate arguments by using the collected non-equivalent method argument lists from the executions of existing tests. This complements existing method parameter generation based on a dedicated test data pool, which contains default data values [9, 26] or user-defined data values [26]. In practice, programmers often write unit tests [3], and these tests often contain some good representative argument values. Our method parameter generation takes advantage of these tests, rather than requiring programmers to explicitly define representative argument values. When there are no manually written tests for a class, we can generate non-equivalent method argument lists based on tests generated by existing test-generation tools.

Our test generation is a type of combinatorial testing. We generate tests to exercise each possible combination of non-equivalent object states and non-equivalent method argument lists. In object state setup, we collect non-equivalent object states from the executions of existing tests. Since these non-equivalent object states might not be exhaustively exercised by all non-equivalent method argument lists, we generate tests to exercise each non-equivalent object state with all non-equivalent method argument lists. After we execute the new generated tests, we might collect some more new non-equivalent object states that are not encountered in the executions of existing tests. Then we can apply our test-generation technique to generate more tests to exercise them in another iteration. The pseudo-code of the test-generation algorithm is presented in Figure 4.

```
Set testgen(Set existingTests, int maxIterNum) {
  Set newTests = new Set();
  RuntimeInfo runtimeInfo = runAndCollect(existingTests);
  Set nonEqArgLists = runtimeInfo.getNonEqArgLists();
  Set frontiers = runtimeInfo.getNonEqObjStates();
  for(int i=1;i<=maxIterNum && frontiers.size()>0;i++) {
      Set newTestsForCurIter = new Set();
      foreach (objState in frontiers) {
        foreach (argList in nonEqArgLists) {
          Test newTest = makeTest(objState, argList);
          newTestsForCurIter.add(newTest);
          newTests.add(newTest);
        }
      }
      runtimeInfo = runAndCollect(newTestsForCurIter);
      frontiers = runtimeInfo.getNonEqObjStates();
  }
  return newTests;
}
```

Figure 4: Pseudo-code implementation of the test-generation algorithm.

Given a set of existing tests and a user-defined maximum iteration number, our test-generation algorithm first runs the existing tests and collect runtime information, including non-equivalent method argument lists and non-equivalent object states. We also collect the method sequence that leads to a non-equivalent object state or an argument in a method argument list. We use these method sequences to reproduce object states or arguments. We put the collected non-equivalent object states into a frontier set. Then we iterate each object state in the frontier set and invoke each non-equivalent method argument list on the object state. Each combination of an object state and a method argument list forms a test. After we generate tests based on all combinations, we run all new tests generated in the current iteration and collect runtime information. We collect new non-equivalent object states that are encountered in the current iteration, and set them as the new frontier set. With this new frontier set, we start the subsequent iteration until we have reached the maximum iteration number or the frontier set has no object state. Then we return the collected generated tests over all iterations. These tests are exported to a test class.

Since invoking a state-preserving method on an object state does not change the state, we can still invoke other methods on the object state in the same test. We merge generated tests as much as possible by reusing and sharing the same object states among multiple method argument lists. This reduces the number of the generated tests and the execution cost of the generated test suite. The generated test suite contains no redundant tests, since our combinatorial generation mechanism guarantees that the last method execution produced by each test is not equivalent to any method execution produced by earlier executed tests.

In our tool implementation, we use Java reflection mechanisms [1] to generate and execute new tests online. In the end of test generation, we export the tests generated after each iteration to a JUnit test class code [21], based on JCrasher's test code generation functionality [9].

5. EXPERIMENTS

This section presents experimental results of our test-minimization and test-generation tools. We hypothesize that our test-minimization tool can reduce a significant number of tests automatically generated by existing test-generation tools, and our test-generation tool can effectively generate non-redundant tests to exercise non-equivalent object states. We conduct two experiments to validate our hypotheses, and compare the effectiveness of different techniques. We perform all experiments on a Linux machine with a Pentium IV 1.1 GHz processor using Sun's Java 2 SDK 1.4.2 JVM with default configurations.

Table 2: Subject programs used in the experiments

program	size	ncnb loc	Jtest		JCrasher	
			tests	mexecs	tests	mexecs
IntStack	4	32	94	348	6	11
UBStack	10	77	1423	14067	14	27
BSet	9	59	1643	12492	45	87
BBag	8	77	1173	5153	90	177
ShoppingCart	7	39	470	1652	31	61
BankAccount	6	28	519	2554	135	261
BinarySearchTree	10	112	1384	6236	36	107
LinkedList	10	67	1965	9272	145	347

5.1 Subjects

We use eight Java classes in our experiments. The `IntStack` class is the running example. The `UBStack`, `BSet`, and `BBag` classes are taken from the experimental subjects used by Stotts et al. [18, 30]. The `ShoppingCart` class is a popular example for using JUnit [8]. The `BankAccount` class is one of the examples distributed with Jtest [26]. The `BinarySearchTree` and `LinkedList` classes are data structures from a textbook [34]. The first three columns of Table 2 show the class name, the number of public method, and the number of non-comment, non-blank lines of code for each subject respectively.

We use two third-party test-generation tools: Jtest [26] and JCrasher [9] to automatically generate test inputs for program subjects. Jtest allows users to set the length of calling sequences in the range of one to three. In our experiment, we set the length of calling sequences as three. Then Jtest first tries all calling sequences of length one followed by all those of length two and three sequentially. JCrasher automatically constructs method sequences to generate non-primitive arguments, and uses default data values for primitive arguments. JCrasher generates tests with the length of calling sequences as one. The last four columns of Table 2 show the number of Jtest-generated tests, their exercised method executions, JCrasher-generated tests, and their exercised method executions respectively.

5.2 Experimental Results

As is discussed in Section 4, the ModifyingSeq technique cannot be used to represent argument object states in our implementation. Our experiments focus on the comparison of using different techniques to represent receiver object states, and use the WholeSeq technique, the most conservative one, to represent argument object states. In the first experiment, we apply our test-minimization tool to minimize tests automatically generated by Jtest and JCrasher. Figure 5 and Figure 8 show the percentage of minimized (redundant) tests among tests generated by Jtest and JCrasher respectively. Figure 6 and Figure 9 show the percentage of minimized method executions among method executions generated by Jtest and JCrasher respectively. We observe that all techniques except for the WholeSeq technique substantially remove over 90% of tests and method executions generated by Jtest for most subjects, and 30% of the ones generated by JCrasher for half of the subjects. The three concrete-state representation techniques minimize more tests than the two method-sequence representation techniques. There is no significant difference in the number of minimized tests by these three concrete-state representation techniques. Figure 7 and Figure 10 show the elapsed real time of test minimization on Jtest-generated and JCrasher-generated tests respectively. The elapsed time does not include the instrumentation time of a test class, which is the same for all techniques, ranging from several seconds to a minute. The three concrete-state representation techniques take longer time than the two method-sequence representation techniques. There is no significant difference in the elapsed time of minimized tests by these three concrete-state representation techniques. We also measure the branch coverage and the number of different uncaught thrown exceptions for the original test suite generated by Jtest or JCrasher and its minimized test suite. The results show that the minimized test suite achieves the same branch coverage and the same number of different uncaught thrown exceptions as the original test suite.

In the second experiment, we use our test-generation tool to augment tests automatically generated by Jtest and JCrasher. We set the maximum iteration number as two. Figure 11 and Figure 12 show the average number (averaging across subjects) of tests and method executions generated by Jtest, JCrasher, and our test-generation tool. The first two bars being marked with `Existing` indicate the average number of tests generated by Jtest and JCrasher respectively. The remaining bars show the average number of tests generated by our test-generation tool based on Jtest-generated or JCrasher-generated tests using different techniques. We do not include the results of using the WholeSeq technique in the figures, since the test generation based on the WholeSeq technique causes out-of-memory exceptions for most subjects in the second iteration. Figure 13 shows the average number of non-equivalent object states exercised by tests generated by Jtest, JCrasher, and our test-generation tool. The bars associated with `Jtest-Existing` and `JCrasher-Existing` indicate the results for the existing tests generated by Jtest and JCrasher respectively. The bars associated with `Jtest-Iteration 2` and `JCrasher-Iteration 2` show the results for the tests generated by our test-generation tool based on Jtest-generated and JCrasher-generated tests respectively. Figure 14 shows the branch coverage percentage of tests generated for each subject by Jtest, JCrasher, and our test-generation tool. Since the branch coverage achieved by tests generated using different techniques are the same, we put different subjects instead of different techniques in the `x` axis. From the results for test generation, we observe that our test-generation tool using our techniques (except for the WholeSeq technique) generate fewer tests than Jtest, and these tests exercise more non-equivalent object states and more branches. In addition, the tests generated by our test-generation tool based on Jtest-generated tests throw two more different uncaught exceptions than the original tests generated by Jtest for the `UBStack` subject, which throw two different uncaught exceptions. Our test-generation tool generates slightly more tests than JCrasher, and these tests also exercise more non-equivalent object states and more branches. The tests generated by our test-generation tool based on JCrasher-generated tests increase the number of different uncaught exceptions from zero to two for the `UBStack` subject, from one to two for the `ShoppingCart` subject, from zero to three for the `BinarySearchTree` subject.

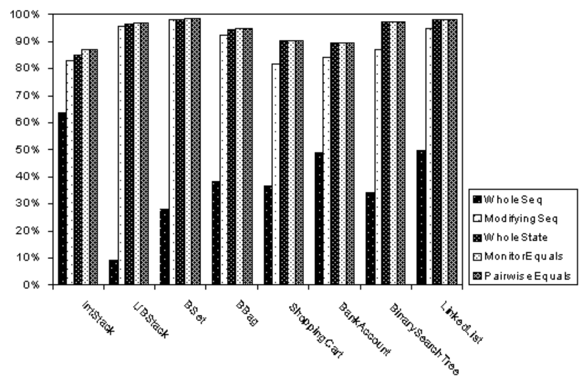

Figure 5: Percentage of redundant tests among Jtest-generated tests

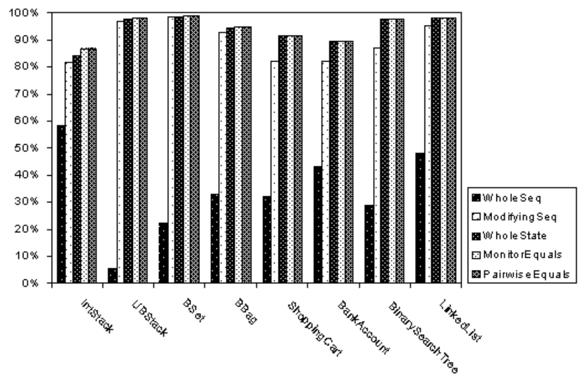

Figure 6: Percentage of minimized method executions among Jtest-generated method executions

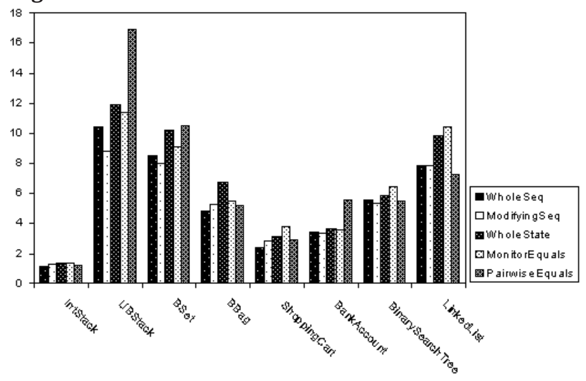

Figure 7: Elapsed real time (in seconds) of test minimization on Jtest-generated tests

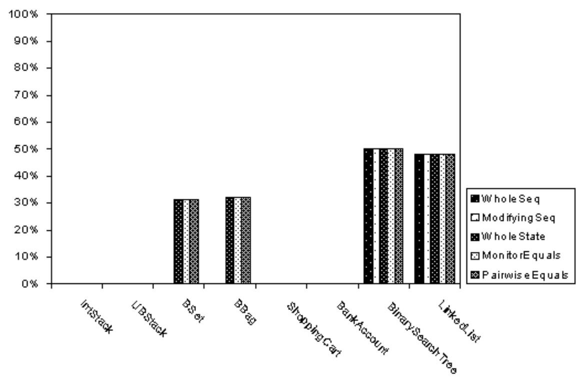

Figure 8: Percentage of redundant tests among JCrasher-generated tests

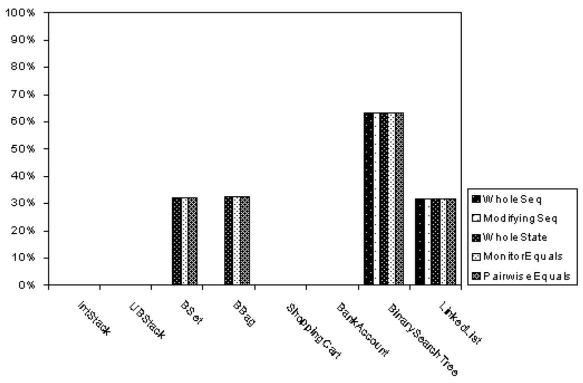

Figure 9: Percentage of minimized method executions among JCrasher-generated method executions

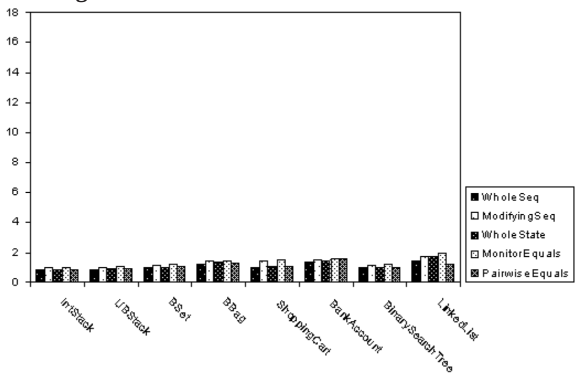

Figure 10: Elapsed real time (in seconds) of test minimization on JCrasher-generated tests

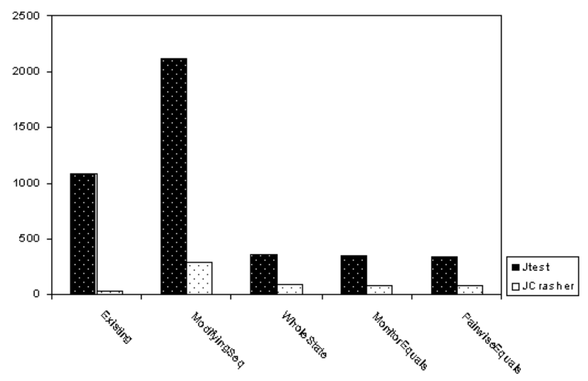

Figure 11: Average number of tests generated by Jtest and JCrasher, and our test-generation tool based on Jtest-generated and JCrasher-generated tests (after 2 iterations)

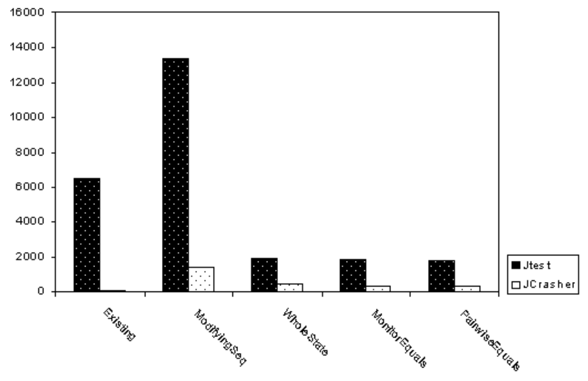

Figure 12: Average number of method executions generated by Jtest and JCrasher, and our test-generation tool based on Jtest-generated and JCrasher-generated tests (after 2 iterations)

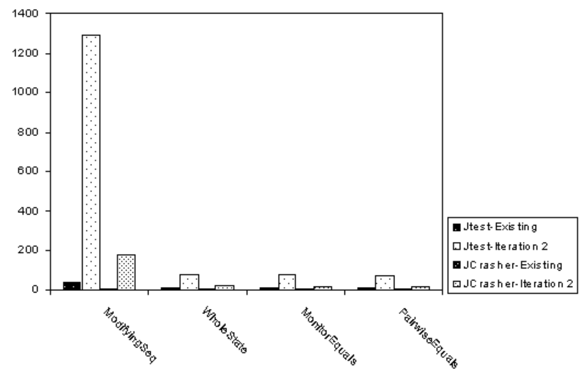

Figure 13: Average number of non-equivalent object states exercised by tests generated by Jtest and JCrasher and our test-generation tool based on Jtest-generated and JCrasher-generated tests (after 2 iterations)

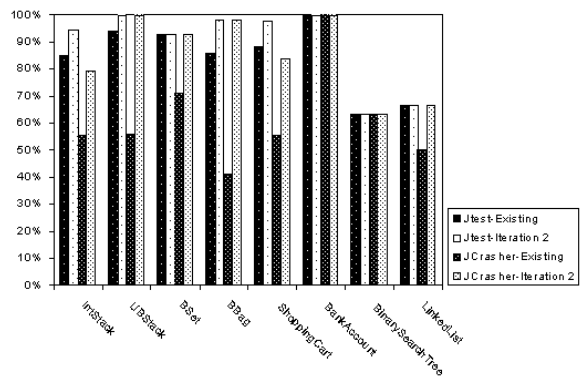

Figure 14: Branch coverage percentage by tests generated by Jtest, JCrasher and our test-generation tool based on Jtest-generated and JCrasher-generated tests (after 2 iterations)

5.3 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. We mainly used data structures as our subject programs. Among two third-party tools, Jtest is one of the testing tools popularly used in industry. These threats could be further reduced by more experiments on wider types of subjects and third-party tools. The main threats to internal validity include instrumentation effects that can bias our results. Faults in our tools, Jtest, or JCrasher might cause such effects. To reduce these threats, we manually inspected the collected execution traces for most program subjects. The main threats to construct validity include the uses of those measurements in our experiments to assess our tools. To assess the effectiveness of our test-generation tool, we mainly measured the number of new non-equivalent object states, different uncaught exceptions, and the percentage of branches that are exercised by new generated tests. In future work, we plan to measure the fault-detecting capability of new generated tests more thoroughly.

6. RELATED WORK

Previous work has developed techniques to detect object state equivalence. Observational equivalence [4, 11, 17] techniques are much more expensive than our techniques. Our techniques sometimes give more conservative results than observational equivalence techniques. The serialize-and-hash technique [17] is similar to our WholeState technique. Most of these previous techniques in detecting object state equivalence are used to verify the correctness of axioms or infer axioms in algebraic specifications. We detect equivalent object states mainly for avoiding redundant tests in test generation.

There are some other research projects in encoding and comparing program or object states. Zimmermann and Zeller develop a memory graph and its visualization to capture and explore program states during C program executions [39]. They reduce the comparison of program states to the comparison of graphs. Zeller's later work compares memory graphs to isolate cause-effect chains of a program failure [38]. Iosif [19] and Robby et al. [27] use a similar linearization technique to encode states in model checkers. They do not apply any technique to collect relevant object fields, but collect all the fields. Our previous work [36] uses the C front end of the Daikon tool [12] to output program states at method entry and exit points. We assemble these program states in different ways to form different levels of value spectra. Then we compare the value spectra from the executions of an old and a new program versions, and use the results to aid regression fault exposure and localization. In future work, we plan to apply our state representation techniques in the value spectra approach for object-oriented programs.

Several lines of previous work generate tests to exercise object states without requiring any specification. Buy et al. use data flow analysis, symbolic execution, and automated deduction to generate method call sequences exercising definition-use pairs of object fields [6]. Our test-generation tool generates method calls to fully exercise non-equivalent object states. Our generated tests for exercising object states implicitly generate method sequences. Ball et al. present an approach for automated testing of container classes based on combinatorial algorithm for state generation [2]. Our test-generation tool applies a similar combinatorial mechanism. Our approach is totally automatic, whereas Ball et al's approach requires a dedicated state generator. Kung et al. propose an object state test model and use symbolic execution to statically extract an abstract model from C++ source code [22]. They use this test model to guide test generation. Our approach dynamically detects equivalent object states and incrementally exercise those new object states.

Given specifications for a program, several other research projects generate tests to exercise object states. Boyapati et al. develop the Korat tool to exhaustively generate valid object states that are bounded by a user-defined size [5]. Korat monitors field accesses within the execution of a Java predicate and uses this information to prune the search for valid test inputs. Our MonitorEquals technique uses the actual values of accessed fields to represent state, and uses isomorphism to compare states for equivalence. Korat's generation also guarantees that the generated objects are non-isomorphic. Grieskamp et al. allow the user to define indistinguishability properties to group infinite states in abstract state machines into equivalence classes, called hyperstates [15]. Their tool incrementally produces finite state machines by executing abstract state machines. Our test generation works in a similar way by incrementally producing new object states.

Whaley et al. dynamically extract Java component interface models, each of which accesses the same field [35]. They statically determine whether a method is a state-modifying one. In their extracted models, they assume that the same state-modifying method transits an object to the same abstract state. This assumption makes the extracted models more compact. Our ModifyingSeq technique dynamically and accurately determines if a particular method call is state-modifying. The object state representations by our techniques are more conservative and accurate than Whaley et al.'s approach. In our ongoing work, we extract a state transition model from test executions based on object state equivalence. In our preliminary experiments, we observe that our extracted models are more complex but more accurate than the models extracted by Whaley et al.'s approach.

Our previous work integrates Daikon [12] and Jtest [26] and uses operational violations to select a small valuable subset of automatically generated tests for inspection [37]. We can view that our new test-minimization technique is trying to conservatively minimize automatically generated tests from the other end: removing useless tests. There are other lines of work in minimizing or prioritizing tests for regression testing [16, 28, 29]. Although changing a program can make a redundant test on the old version not redundant any more, we can apply regression test prioritization techniques based on non-equivalent object state coverage. In addition, if two method sequences on the old version produce equivalent object states, and the modifications do not affect the executions of these two method sequences, we can still safely determine the object states resulting from these two sequences on the new version are equivalent. In future work, we plan to investigate the application of our techniques in regression testing.

7. CONCLUSION

We have proposed a framework for detecting redundant tests based on equivalent objects and presented five techniques within this framework. We have also presented our test-minimization and test-generation tools based on these techniques. Our tools produce only non-redundant tests. We have conducted experiments to assess the effectiveness of minimizing and augmenting tests generated by two third-party test-generation tools. The results show that we can substantially reduce the size of test suites generated by these tools, and we can effectively generate tests to augment these test suites to exercise more non-equivalent object states. These results strongly suggest that tools and techniques for generation of object-oriented test suites must consider avoiding redundant tests.

8. REFERENCES

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

[2] T. Ball, D. Hoffman, F. Ruskey, R. Webber, and L. J. White. State generation and automated class testing. *Software Testing, Verification and Reliability*, 10(3):149–170, 2000.

[3] K. Beck. *Test Driven Development: By Example*. Addison-Wesley, 2003.

[4] 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.

[5] 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. ACM Press, 2002.

[6] U. Buy, A. Orso, and M. Pezze. Automated testing of classes. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 39–48. ACM Press, 2000.

[7] 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)*, June 2002.

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

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

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

[11] 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.

[12] 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.

[13] Foundations of Software Engineering, Microsoft Research. The AsmL test generator tool. `http://research.microsoft.com/fse/asml/doc/AsmLTester.html`.

[14] E. Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, 1960.

[15] 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. ACM Press, 2002.

[16] 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. IEEE Computer Society, 2003.

[17] J. Henkel and A. Diwan. Discovering algebraic specifications from java classes. In L. Cardelli, editor, *17th European Conference on Object-Oriented Programming*, pages 431–456, Darmstadt, Germany, 2003. Springer.

[18] M. Hughes and D. Stotts. Daistish: systematic algebraic testing for oo programs in the presence of side-effects. In *Proceedings of the 1996 international symposium on Software testing and analysis*, pages 53–61. ACM Press, 1996.

[19] R. Iosif. Symmetry reduction criteria for software model checking. In *Proceedings of the 9th SPIN Workshop on Software Model Checking*, volume 2318 of *LNCS*, pages 22–41. Springer, July 2002.

[20] D. S. Johnson. Approximation algorithms for combinatorial problems. *J. Comput. System Sci.*, 9:256–278, 1974.

[21] JUnit. http://www.junit.org.

[22] D. Kung, N. Suchak, J. Gao, and P. Hsia. On object state testing. In *Proceedings of Computer Software and Applications Conference (COMPSAC94)*, pages 222–227. IEEE Computer Society Press, 1994.

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

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

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

[27] Robby, M. B. Dwyer, J. Hatcliff, and R. Iosif. Space-reduction strategies for model checking dynamic systems. In *Proceedings of the 2003 Workshop on Software Model Checking*, July 2003.

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

[29] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *Proceedings of the international symposium on Software testing and analysis*, pages 97–106. ACM Press, 2002.

[30] 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.

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

[32] the AspectJ Team. The aspectj programming guide. Online manual.

[33] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proc. 15th IEEE International Conference on Automated Software Engineering (ASE)*, Grenoble, France, 2000.

[34] M. A. Weiss. *Data Structures and Algorithm Analysis in Java*. Addison Wesley, 1999.

[35] J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *Proceedings of the international symposium on Software testing and analysis*, pages 218–228. ACM Press, 2002.

[36] T. Xie and D. Notkin. Checking inside the black box: Regression fault exposure and localization based on value spectra differences. Technical Report UW-CSE-02-12-04, University of Washington Department of Computer Science and Engineering, Seattle, WA, December 2002.

[37] 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. IEEE Computer Society, 2003.

[38] A. Zeller. Isolating cause-effect chains from computer programs. *SIGSOFT Softw. Eng. Notes*, 27(6):1–10, 2002.

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

9. APPENDIX

This appendix presents detailed results for all experiments. Discussion of these results is in Section 5.

Table 3 in the appendix shows the actual numbers of the test minimization results. The `%r-tests` column shows the percentage of minimized (redundant) tests among all generated tests; the `%m-mexecs` column shows the percentage of minimized (removed) method executions among all method executions executed by generated tests; the `time(sec)` column shows the elapsed real time in seconds spent on the minimization.

Table 4 and 6 in the appendix show the actual numbers of the test generation results based on Jtest-generated tests and JCrasher-generated tests respectively. The `orig`, `i-1`, and `i-2` columns show the data for the Jtest/JCrasher-generated tests, the generated tests in the first iteration, and the generated tests in the second iteration respectively. The `#tests` column shows the number of generated tests; the `#mexecs` column shows the number of generated method executions; the `#neobjs` column shows the number of exercised non-equivalent object states; the `time(sec)` shows the elapsed real time in seconds spent on the generation. The elapsed time does not include the execution time of existing tests, which is roughly equal to the test minimization time in Table 3. When we apply a technique on an iteration and encounter an out-of-memory exception, we put an `om` in the corresponding entry. We set the time out for each iteration as one minute, and if an iteration is time out, we put a `*` before the corresponding data entry. Table 5 and 7 in the appendix show the numbers of different thrown uncaught exceptions and the branch coverage percentage of the generated tests based on Jtest-generated tests and JCrasher-generated tests respectively. The `#exceptions` column shows the number of different thrown uncaught exceptions; the `%bcov` column shows the branch coverage percentage.

Table 3: Experimental results for test minimization

subject	technique	Jtest generated tests			JCrasher generated tests		
		%r-tests	%m-mexecs	time (sec)	%r-tests	%m-mexecs	time (sec)
`IntStack`	WholeSeq	63.8%	58.6%	1.13	0.0%	0.0%	0.80
	ModifyingSeq	83.0%	81.6%	1.25	0.0%	0.0%	1.02
	WholeState	85.1%	84.2%	1.37	0.0%	0.0%	0.81
	MonitorEquals	87.2%	86.8%	1.33	0.0%	0.0%	1.01
	PairwiseEquals	87.2%	86.8%	1.24	0.0%	0.0%	0.84
`UBStack`	WholeSeq	9.1%	5.6%	10.40	0.0%	0.0%	0.84
	ModifyingSeq	95.9%	96.9%	8.82	0.0%	0.0%	1.04
	WholeState	96.6%	97.7%	11.94	0.0%	0.0%	0.87
	MonitorEquals	96.9%	98.1%	11.34	0.0%	0.0%	1.06
	PairwiseEquals	96.9%	98.1%	16.94	0.0%	0.0%	0.91
`BSet`	WholeSeq	27.9%	22.3%	8.52	0.0%	0.0%	0.94
	ModifyingSeq	98.3%	98.5%	7.99	0.0%	0.0%	1.14
	WholeState	98.3%	98.5%	10.19	31.1%	32.2%	1.02
	MonitorEquals	98.5%	98.8%	9.13	31.1%	32.2%	1.19
	PairwiseEquals	98.5%	98.8%	10.55	31.1%	32.2%	1.06
`BBag`	WholeSeq	38.4%	32.9%	4.87	0.0%	0.0%	1.20
	ModifyingSeq	92.7%	92.8%	5.30	0.0%	0.0%	1.43
	WholeState	94.2%	94.5%	6.78	32.2%	32.8%	1.35
	MonitorEquals	94.7%	95.1%	5.52	32.2%	32.8%	1.47
	PairwiseEquals	94.7%	95.1%	5.15	32.2%	32.8%	1.30
`ShoppingCart`	WholeSeq	36.6%	32.2%	2.41	0.0%	0.0%	0.99
	ModifyingSeq	81.9%	82.2%	2.86	0.0%	0.0%	1.43
	WholeState	90.6%	91.6%	3.15	0.0%	0.0%	1.10
	MonitorEquals	90.6%	91.6%	3.83	0.0%	0.0%	1.54
	PairwiseEquals	90.6%	91.6%	2.93	0.0%	0.0%	1.07
`BankAccount`	WholeSeq	48.9%	43.1%	3.46	0.0%	0.0%	1.34
	ModifyingSeq	84.0%	82.4%	3.40	0.0%	0.0%	1.53
	WholeState	89.8%	89.3%	3.65	0.0%	0.0%	1.40
	MonitorEquals	89.8%	89.3%	3.56	0.0%	0.0%	1.60
	PairwiseEquals	89.8%	89.3%	5.53	0.0%	0.0%	1.62
`BinarySearchTree`	WholeSeq	33.9%	28.8%	5.57	50.0%	63.6%	0.96
	ModifyingSeq	87.4%	87.1%	5.33	50.0%	63.6%	1.17
	WholeState	97.4%	97.8%	5.86	50.0%	63.6%	1.00
	MonitorEquals	97.4%	97.8%	6.47	50.0%	63.6%	1.22
	PairwiseEquals	97.4%	97.8%	5.47	50.0%	63.6%	1.04
`LinkedList`	WholeSeq	49.5%	48.3%	7.85	48.3%	31.4%	1.50
	ModifyingSeq	95.0%	95.3%	7.83	48.3%	31.4%	1.71
	WholeState	98.1%	98.4%	9.82	48.3%	31.4%	1.75
	MonitorEquals	98.1%	98.4%	10.40	48.3%	31.4%	1.91
	PairwiseEquals	98.1%	98.4%	7.28	48.3%	31.4%	1.22

Table 4: Experimental results (1) for test generation based on Jtest generated tests

		#tests			#mexecs			#neobjs			time(sec)	
subject	technique	orig	i-1	i-2	orig	i-1	i-2	orig	i-1	i-2	i-1	i-2
`IntStack`	WholeSeq	94	270	1005	348	1500	6786	59	217	805	1.00	3.03
	ModifyingSeq	94	52	148	348	257	913	13	37	105	0.28	0.81
	WholeState	94	36	76	348	166	436	9	19	40	0.25	0.54
	MonitorEquals	94	28	60	348	123	325	7	15	31	0.22	0.45
	PairwiseEquals	94	28	60	348	123	325	7	15	31	0.17	0.39
`UBStack`	WholeSeq	1423	om	om	14067	om	om	8718	om	om	om	om
	ModifyingSeq	1423	85	265	14067	539	1955	17	53	170	0.35	0.95
	WholeState	1423	45	93	14067	261	576	9	17	24	0.25	0.64
	MonitorEquals	1423	35	71	14067	193	417	7	13	15	0.19	0.48
	PairwiseEquals	1423	35	71	14067	193	417	7	13	15	0.24	0.50
`BSet`	WholeSeq	1643	om	om	12492	om	om	6047	om	om	om	om
	ModifyingSeq	1643	76	188	12492	307	923	10	26	64	0.33	0.68
	WholeState	1643	68	131	12492	257	550	9	18	34	0.38	0.63
	MonitorEquals	1643	60	123	12492	216	509	8	17	27	0.31	0.59
	PairwiseEquals	1643	60	123	12492	216	509	8	17	20	0.36	0.78
`BBag`	WholeSeq	1173	om	om	5153	om	om	1429	om	om	om	om
	ModifyingSeq	1173	593	4064	5153	2938	24936	43	310	2534	2.60	19.33
	WholeState	1173	271	1012	5153	1166	5342	20	77	272	1.59	5.99
	MonitorEquals	1173	257	985	5153	1102	5218	19	75	250	1.39	5.88
	PairwiseEquals	1173	257	985	5153	1102	5218	19	75	231	1.16	7.09
`ShoppingCart`	WholeSeq	470	3185	om	1652	15806	om	306	2941	om	8.15	om
	ModifyingSeq	470	476	2279	1652	2233	13070	36	171	899	2.11	9.67
	WholeState	470	147	501	1652	597	2551	11	37	116	1.15	3.92
	MonitorEquals	470	147	501	1652	597	2551	11	37	116	1.54	6.72
	PairwiseEquals	470	144	410	1652	577	2033	10	31	94	0.98	4.13
`BankAccount`	WholeSeq	519	om	om	2554	om	om	715	om	om	om	om
	ModifyingSeq	519	800	4820	2554	4004	29348	80	482	2894	3.17	21.75
	WholeState	519	290	970	2554	1320	5258	29	97	155	1.36	4.09
	MonitorEquals	519	290	970	2554	1320	5258	29	97	155	1.35	4.04
	PairwiseEquals	519	290	970	2554	1320	5258	29	97	155	4.33	19.99
`BinarySearchTree`	WholeSeq	1384	om	om	6236	om	om	1816	om	om	om	om
	ModifyingSeq	1384	657	4100	6236	3721	27671	78	481	2864	2.34	14.26
	WholeState	1384	58	98	6236	213	416	5	9	10	0.38	0.65
	MonitorEquals	1384	58	98	6236	213	416	5	9	10	0.43	0.81
	PairwiseEquals	1384	58	98	6236	213	416	5	9	10	0.19	0.48
`LinkedList`	WholeSeq	1965	om	om	9272	om	om	1966	om	om	om	om
	ModifyingSeq	1965	217	1092	9272	1335	7940	31	156	781	1.38	6.29
	WholeState	1965	24	30	9272	138	190	4	5	6	0.34	0.45
	MonitorEquals	1965	24	30	9272	138	190	4	5	6	0.34	0.45
	PairwiseEquals	1965	24	30	9272	138	190	4	5	6	0.16	0.25

Table 5: Experimental results (2) for test generation based on Jtest generated tests

		#exceptions			%bcov		
subject	technique	orig	i-1	i-2	orig	i-1	i-2
`IntStack`	WholeSeq	1	1	1	66.7%	66.7%	66.7%
	ModifyingSeq	1	1	1	66.7%	66.7%	66.7%
	WholeState	1	1	1	66.7%	66.7%	66.7%
	MonitorEquals	1	1	1	66.7%	66.7%	66.7%
	PairwiseEquals	1	1	1	66.7%	66.7%	66.7%
`UBStack`	WholeSeq	2	om	om	93.8%	om	om
	ModifyingSeq	2	3	4	93.8%	100.0%	100.0%
	WholeState	2	3	4	93.8%	100.0%	100.0%
	MonitorEquals	2	3	4	93.8%	100.0%	100.0%
	PairwiseEquals	2	3	4	93.8%	100.0%	100.0%
`BSet`	WholeSeq	0	om	om	88.4%	om	om
	ModifyingSeq	0	0	0	88.4%	93.0%	97.7%
	WholeState	0	0	0	88.4%	93.0%	97.7%
	MonitorEquals	0	0	0	88.4%	93.0%	97.7%
	PairwiseEquals	0	0	0	88.4%	93.0%	97.7%
`BBag`	WholeSeq	0	om	om	85.2%	om	om
	ModifyingSeq	0	0	0	85.2%	88.9%	94.4%
	WholeState	0	0	0	85.2%	88.9%	94.4%
	MonitorEquals	0	0	0	85.2%	88.9%	94.4%
	PairwiseEquals	0	0	0	85.2%	88.9%	94.4%
`ShoppingCart`	WholeSeq	2	2	2	92.9%	92.9%	92.9%
	ModifyingSeq	2	2	2	92.9%	92.9%	92.9%
	WholeState	2	2	2	92.9%	92.9%	92.9%
	MonitorEquals	2	2	2	92.9%	92.9%	92.9%
	PairwiseEquals	2	2	2	92.9%	92.9%	92.9%
`BankAccount`	WholeSeq	3	om	om	100.0%	om	om
	ModifyingSeq	3	3	3	100.0%	100.0%	100.0%
	WholeState	3	3	3	100.0%	100.0%	100.0%
	MonitorEquals	3	3	3	100.0%	100.0%	100.0%
	PairwiseEquals	3	3	3	100.0%	100.0%	100.0%
`BinarySearchTree`	WholeSeq	3	om	om	85.7%	om	om
	ModifyingSeq	3	3	3	85.7%	91.1%	98.2%
	WholeState	3	3	3	85.7%	91.1%	98.2%
	MonitorEquals	3	3	3	85.7%	91.1%	98.2%
	PairwiseEquals	3	3	3	85.7%	91.1%	98.2%
`LinkedList`	WholeSeq	1	om	om	63.3%	om	om
	ModifyingSeq	1	1	1	63.3%	63.3%	63.3%
	WholeState	1	1	1	63.3%	63.3%	63.3%
	MonitorEquals	1	1	1	63.3%	63.3%	63.3%
	PairwiseEquals	1	1	1	63.3%	63.3%	63.3%

Table 6: Experimental results (1) for test generation based on JCrasher generated tests

		#tests			#mexecs			#neobjs			time(sec)	
subject	technique	orig	i-1	i-2	orig	i-1	i-2	orig	i-1	i-2	i-1	i-2
IntStack	WholeSeq	6	30	144	11	88	582	6	26	121	0.13	0.58
	ModifyingSeq	6	20	80	11	62	338	4	16	61	0.11	0.41
	WholeState	6	20	75	11	62	315	4	15	48	0.16	0.62
	MonitorEquals	6	20	65	11	62	269	4	13	40	0.16	0.44
	PairwiseEquals	6	20	65	11	62	269	4	13	40	0.12	0.46
UBStack	WholeSeq	14	196	2520	27	587	10215	14	183	2341	0.63	4.67
	ModifyingSeq	14	25	90	27	119	522	5	18	61	0.19	0.69
	WholeState	14	25	70	27	119	398	5	14	21	0.26	0.78
	MonitorEquals	14	25	60	27	119	336	5	12	13	0.26	0.62
	PairwiseEquals	14	25	60	27	119	336	5	12	13	0.34	0.72
BSet	WholeSeq	45	675	9495	87	2022	38478	45	633	8865	1.62	18.27
	ModifyingSeq	45	30	63	87	141	303	6	9	18	0.23	0.43
	WholeState	45	26	48	87	122	230	5	7	7	0.28	0.52
	MonitorEquals	45	26	26	87	122	122	5	5	5	0.27	0.27
	PairwiseEquals	45	26	26	87	122	122	5	5	5	0.30	0.30
BBag	WholeSeq	90	2700	om	177	8097	om	90	2613	om	4.70	om
	ModifyingSeq	90	93	357	177	441	1725	9	21	93	0.61	1.61
	WholeState	90	47	113	177	233	554	5	8	8	0.60	1.08
	MonitorEquals	90	47	47	177	233	233	5	5	5	0.48	0.48
	PairwiseEquals	90	47	47	177	233	233	5	5	5	0.38	0.38
ShoppingCart	WholeSeq	31	558	om	61	1660	om	31	541	om	1.96	om
	ModifyingSeq	31	421	*7909	61	1293	*32129	15	248	*3686	1.77	64.50
	WholeState	31	393	om	61	1205	om	14	184	om	2.53	om
	MonitorEquals	31	169	1166	61	501	4610	6	32	162	1.46	10.87
	PairwiseEquals	31	168	1127	61	500	4452	6	31	156	1.25	14.04
BankAccount	WholeSeq	135	1620	om	261	4824	om	135	1521	om	3.68	om
	ModifyingSeq	135	1170	om	261	3654	om	90	819	om	3.24	om
	WholeState	135	264	495	261	804	1854	24	45	72	1.11	2.00
	MonitorEquals	135	264	495	261	804	1854	24	45	72	1.07	2.06
	PairwiseEquals	135	264	495	261	804	1854	24	45	72	1.69	4.82
BinarySearchTree	WholeSeq	36	324	5310	107	971	21469	18	307	5016	1.09	11.31
	ModifyingSeq	36	104	875	107	387	4160	10	83	658	0.65	3.45
	WholeState	36	43	169	107	175	760	5	14	25	0.43	1.55
	MonitorEquals	36	43	169	107	175	760	5	14	25	0.55	1.83
	PairwiseEquals	36	43	169	107	175	760	5	14	25	0.36	1.10
LinkedList	WholeSeq	145	5402	om	347	28190	om	73	5257	om	15.88	om
	ModifyingSeq	145	3422	om	347	20690	om	58	3307	om	12.62	om
	WholeState	145	116	182	347	487	851	2	3	4	0.72	1.19
	MonitorEquals	145	116	182	347	487	851	2	3	4	0.74	1.14
	PairwiseEquals	145	116	182	347	487	851	2	3	4	0.60	0.89

Table 7: Experimental results (2) for test generation based on JCrasher generated tests

		#exceptions			%bcov		
subject	technique	orig	i-1	i-2	orig	i-1	i-2
`IntStack`	WholeSeq	1	1	1	50.0%	66.7%	66.7%
	ModifyingSeq	1	1	1	50.0%	66.7%	66.7%
	WholeState	1	1	1	50.0%	66.7%	66.7%
	MonitorEquals	1	1	1	50.0%	66.7%	66.7%
	PairwiseEquals	1	1	1	50.0%	66.7%	66.7%
`UBStack`	WholeSeq	0	1	2	56.2%	90.6%	100.0%
	ModifyingSeq	0	1	2	56.2%	90.6%	100.0%
	WholeState	0	1	2	56.2%	90.6%	100.0%
	MonitorEquals	0	1	2	56.2%	90.6%	100.0%
	PairwiseEquals	0	1	2	56.2%	90.6%	100.0%
`BSet`	WholeSeq	0	0	0	55.8%	83.7%	83.7%
	ModifyingSeq	0	0	0	55.8%	83.7%	83.7%
	WholeState	0	0	0	55.8%	83.7%	83.7%
	MonitorEquals	0	0	0	55.8%	83.7%	83.7%
	PairwiseEquals	0	0	0	55.8%	83.7%	83.7%
`BBag`	WholeSeq	0	0	0	55.6%	79.6%	79.6%
	ModifyingSeq	0	0	0	55.6%	79.6%	79.6%
	WholeState	0	0	0	55.6%	79.6%	79.6%
	MonitorEquals	0	0	0	55.6%	79.6%	79.6%
	PairwiseEquals	0	0	0	55.6%	79.6%	79.6%
`ShoppingCart`	WholeSeq	1	2	2	71.4%	92.9%	92.9%
	ModifyingSeq	1	2	2	71.4%	92.9%	92.9%
	WholeState	1	2	2	71.4%	92.9%	92.9%
	MonitorEquals	1	2	2	71.4%	92.9%	92.9%
	PairwiseEquals	1	2	2	71.4%	92.9%	92.9%
`BankAccount`	WholeSeq	3	3	3	100.0%	100.0%	100.0%
	ModifyingSeq	3	3	3	100.0%	100.0%	100.0%
	WholeState	3	3	3	100.0%	100.0%	100.0%
	MonitorEquals	3	3	3	100.0%	100.0%	100.0%
	PairwiseEquals	3	3	3	100.0%	100.0%	100.0%
`BinarySearchTree`	WholeSeq	0	3	3	41.1%	89.3%	98.2%
	ModifyingSeq	0	3	3	41.1%	89.3%	98.2%
	WholeState	0	3	3	41.1%	89.3%	98.2%
	MonitorEquals	0	3	3	41.1%	89.3%	98.2%
	PairwiseEquals	0	3	3	41.1%	89.3%	98.2%
`LinkedList`	WholeSeq	1	1	1	63.3%	63.3%	63.3%
	ModifyingSeq	1	1	1	63.3%	63.3%	63.3%
	WholeState	1	1	1	63.3%	63.3%	63.3%
	MonitorEquals	1	1	1	63.3%	63.3%	63.3%
	PairwiseEquals	1	1	1	63.3%	63.3%	63.3%