Detecting Redundant Unit Tests for AspectJ Programs

Tao Xie[1] Jianjun Zhao[2] Darko Marinov[3] David Notkin[4]

[1] Department of Computer Science, North Carolina State University, USA

[2] Department of Computer Science & Engineering, Shanghai Jiao Tong University, China

[3] Department of Computer Science, University of Illinois at Urbana-Champaign, USA

[4] Department of Computer Science & Engineering, University of Washington, USA

xie@csc.ncsu.edu, zhao-jj@cs.sjtu.edu.cn, marinov@cs.uiuc.edu, notkin@cs.washington.edu

Abstract

Aspect-oriented software development is gaining popularity with the adoption of languages such as AspectJ. Testing is an important part in any software development, including aspect-oriented development. To automate generation of unit tests for AspectJ programs, we can apply the existing tools that automate generation of unit tests for Java programs. However, these tools can generate a large number of test inputs, and manually inspecting the behavior of the software on all these inputs is time consuming. We propose Raspect, a framework for detecting redundant unit tests for AspectJ programs. We introduce three levels of units in AspectJ programs: advised methods, advice, and intertype methods. We show how to detect at each level redundant tests that do not exercise new behavior. Our approach selects only non-redundant tests from the automatically generated test suites, thus allowing the developer to spend less time in inspecting this reduced set of tests. We have implemented Raspect and applied it on 12 subjects taken from a variety of sources; our experience shows that Raspect can effectively reduce the size of generated test suites for inspecting AspectJ programs.

1 Introduction

Aspect-oriented software development (AOSD) is a new paradigm that supports separation of concerns in software development [4, 13, 16, 21]. AOSD makes it possible to modularize crosscutting aspects of a software system. The research in AOSD has so far focused primarily on problem analysis, software design, and implementation activities.

Little attention has been paid to testing in AOSD, although it is well known that testing is a labor-intensive process that can account for half the total cost of software development [3]. Automated software testing, and in particular test generation, can significantly reduce this cost. Although AOSD can lead to better-quality software, AOSD by itself does not guarantee correct software. An aspect-oriented design can lead to a better system architecture, and an aspect-oriented programming language enforces a disciplined coding style, but they do not protect against programmers mistakes. As a result, software testing remains an important task in AOSD.

Aspect-oriented programming languages, such as AspectJ [13], introduce some new language constructs–most notably aspects, advice, and join points–to the common object-oriented programming languages such as Java. These specific constructs require adapting the common testing concepts for testing aspect-oriented software.

We focus on *unit testing*, the process of testing each basic component (a unit) of a program to validate that it correctly implements its detailed design. For aspect-oriented programs, the basic testing unit can be a part of an aspect or a class. In unit testing, developers isolate the unit to run independently from its environment, which allows writing small testing code to exercise the unit alone. However, in aspect-oriented programming, it is unusual to run an aspect in isolation. After all, the intended use of an aspect is to affect the behavior of one or more classes through join points and advice. Thus, the aspects are usually tested in the context with some affected classes, which also allows testing the complex interactions between the aspect and the affected classes.

A part of our previous work [23] leverages the existing tools that automate test generation for Java to automate test generation for the aspects and their affected classes. Test-generation tools for Java are available commercially (e.g., Jtest [18]) or as research prototypes (e.g., JCrasher [5]). These tools test a class by generating and executing numerous method sequences on the objects of the class. Because typical programs do not have executable specifications for automatic correctness checking, these tools rely on developers to inspect the executions of the generated tests for correctness.

Another part of our previous work [22] proposed the Rostra framework for detecting *redundant* tests in Java programs, i.e., tests that do not exercise new behavior of the Java classes under test. Using Rostra, we found that automatic test-generation tools may generate a large number of such tests [22], which only increases the testing time, without increasing the ability to detect faults. Redundant tests are even more common in testing aspects: the tests that differ for the affected class can often be the same for the aspect. (The reverse can also happen, but much more infrequently.) Thus, it is important to avoid redundant tests, not only to reduce the time on test generation and execution, but also to reduce the time that developers need to spend inspecting the tests.

This paper makes the following contributions:

- We propose Raspect, a novel framework for detecting redundant unit tests for AspectJ programs; to the best of our knowledge, this is the first such framework. Raspect extends our previous Rostra framework [22], which detects redundant tests for Java methods. For Raspect, we extend the definitions of redundant tests and the implementation to detect redundant tests for advice and intertype methods.
- We present an implementation of Raspect for detecting redundant unit tests for advised methods, advice, and intertype methods.
- We evaluate Raspect on 12 AspectJ programs from a variety of sources. The results show that Raspect can effectively reduce the size of generated test suites for inspecting AspectJ program behavior.

2 AspectJ

AspectJ is a seamless, aspect-oriented extension to Java. AspectJ adds to Java several new constructs, including join points, pointcuts, advice, intertype declarations, and aspects. An *aspect* is a modular unit of crosscutting implementation in AspectJ. Each aspect encapsulates functionality that may crosscut several classes in a program. Like a class, an aspect can be instantiated, can contain state and methods, and can be specialized with sub-aspects. An aspect is composed with the classes it crosscuts according to the descriptions given in the aspect.

A central concept in the composition of an aspect with other classes is a *join point*. A join point is a well-defined point in the execution of a program, such as a call to a method, an access to an attribute, an object initialization, or an exception handler. Sets of join points may be represented by *pointcuts*, implying that they crosscut the system.

An aspect can specify a piece of *advice* that defines the code that should be executed when the executions reach a join point. Advice is a method-like mechanism, which consists of instructions that execute *before*, *after*, or *around* a join point. The `around` advice executes *in place* of the indicated join point, which allows the aspect to replace a method. An aspect can also use an *intertype declaration* to add a public or private method, field, or interface implementation declaration into a class.

An AspectJ compiler ensures that the base and aspect code run together properly interleaved [1, 11]. The compiler uses *aspect weaving* to compose the code of the *base class* (to which an aspect is applied) and the aspect to ensure that the applicable advice runs at the appropriate join points. After aspect weaving, the base classes are called *woven classes*, and their methods are called *advised methods*.

We next present briefly how the AspectJ compiler translates aspects, advice, and intertype methods; more details of AspectJ are available elsewhere [1].

The AspectJ compiler translates each aspect into a standard Java class (called *aspect class*) and each piece of advice declared in the aspect into a public non-static method in the aspect class. The parameters of this public method are the same as the parameters of the advice, possibly in addition to some `thisJoinPoint` parameters that represent the information about the join point. The body of this public method is usually the same as the body of the advice. At appropriate locations of the base class, the AspectJ compiler inserts calls to the advice. At each site of these inserted calls, first a singleton object of an aspect class is obtained by calling the static method `aspectOf` defined in the aspect class, and then a piece of advice is invoked on the aspect object.

Both `before` and `after` pieces of advice are compiled into public methods of an aspect class in the preceding way; however, compiling and weaving `around` advice is more complicated. Normally a piece of `around` advice is also compiled into a public method in the aspect class. But it takes one additional argument: an `AroundClosure` object. A call to `proceed` in the compiled `around` advice body is replaced with a call to a `run` method on the `AroundClosure` object. However, when an `AroundClosure` object is not needed, the `around` advice is inlined in the base class as a static private method whose first argument is an object of the base class, being the receiver object of the advised method at runtime.

The AspectJ compiler translates each intertype method declaration into a public static method (called *intertype method*) in the aspect class. The parameters of this public method are the same as the parameters of the declared method in the aspect, except that the declared method's receiver object is inserted as the first parameter of the intertype method. The body of this public method is usually the same as the body of the declared method. The AspectJ compiler inserts a wrapper method in the base class and

```
class Cell {
  int data;  Cell next;
  Cell(Cell n, int i) { next = n; data = i; }
}

public class Stack {
  Cell head;
  public Stack() { head = null; }
  public boolean push(int i) {
     if (i < 0) return false;
     head = new Cell(head, i);
     return true;
  }
  public int pop() {
     if (head == null)
       throw new RuntimeException("empty");
     int result = head.data; head = head.next;
     return result;
  }
  Iterator iterator() { return new StackItr(head); }
}
```

Figure 1. An integer stack implementation

```
interface Iterator {
   public boolean hasNext();
   public int next();
}

public class StackItr implements Iterator {
   private Cell cell;
   public StackItr(Cell head) { this.cell = head; }
   public boolean hasNext() { return cell != null; }
   public int next() {
     int result = cell.data;
     cell = cell.next;
     return result;
   }
}
```

Figure 2. Stack iterator

this wrapper method invokes the actual method implementation in the aspect class. The AspectJ compiler compiles each intertype field declaration into a field in the base class. However, all accesses to the fields inserted in the base class are through two public static wrapper methods in the aspect class: one for getting field and the other for setting field. (The compiler also adds the "`get`" and "`set`" methods for the intertype fields in the aspect.)

3 Example

We next illustrate how Raspect determines redundant tests for an AspectJ program. We use a simple integer stack example adapted from Rinard et al. [19]. Figure 1 shows the stack implementation. Objects of the `Cell` class store and link the stack items. The `Stack` class has two public non-constructor methods, `push` and `pop` that implement the standard stack operations, and one package-private method, `iterator` that returns an iterator used to traverse the items in the stack. Figure 2 shows the implementation of the iterator class. Figure 3 shows four aspects for the stack class: `NonNegative`, `NonNegativeArg`, `Instrumentation`, and `PushCount`.

```
aspect NonNegative {
  before(Stack stack) : call(* Stack.*(..)) &&
              target(stack) &&
             !within(NonNegative) {
   Iterator it = stack.iterator();
   while (it.hasNext()) {
     int i = it.next();
     if (i < 0) throw new RuntimeException("negative");
   }
  }
}

aspect NonNegativeArg {
  before() : execution(* Stack.*(..)) {
    Object args[] = thisJoinPoint.getArgs();
    for(int i=0; i<args.length; i++) {
      if ((args[i] instanceof Integer) &&
       (((Integer)args[i]).intValue() < 0))
         System.err.println("Negative argument of " +
              thisJoinPoint.getSignature().toShortString());
    }
  }
}

aspect Instrumentation {
  static int count = 0;
  public print() { System.out.println("count = " + count); }
  after() : call(Stack.new()) { count = 0; }
  after() : call(* Stack.push(int)) { count++; }
}

aspect PushCount {
  int Stack.count = 0;
  public void Stack.increaseCount() { count++; }
  boolean around(Stack stack) :
    execution(* Stack.push(int)) && target(stack) {
      boolean ret = proceed(stack);
      stack.incrementCount();
      return ret;
  }
}
```

Figure 3. Four aspects for stack

This stack implementation accommodates only nonnegative integers as stack items. The `NonNegative` aspect checks this property: the aspect contains a piece of advice that iterates through all items to check whether they are nonnegative. The advice executes before a call to any `Stack` method. The `NonNegativeArg` aspect checks whether `Stack` method arguments are nonnegative. The aspect contains a piece of advice that checks all arguments of an about to be executed `Stack` method. The advice executes before a call to any `Stack` method.

The `Instrumentation` aspect counts the number of times the `push` method is invoked on a stack object since its creation.[1] The aspect contains a piece of advice that increases the static `count` field defined in the aspect. The advice is executed after any call to the `push` method. The aspect contains another piece of advice that resets the static `count` field. This piece of advice is executed after any call to the `Stack` constructor.

The `PushCount` aspect is another version for counting the number of times `push` is invoked on an object since its creation. The aspect contains one intertype declara-

[1] The advice implementation works correctly only when there is no interleaving among `push` and constructor calls.

tion that declares a `count` field for the `Stack` class. The field records the number of times `push` is invoked. The aspect contains another intertype declaration that declares an `increaseCount` method for the class.[2] The method increases the `count` intertype field of `Stack`. The aspect also contains a piece of around advice that invokes the `increaseCount` intertype method declared in the aspect. The advice is executed around any execution of `push`.

After we use the AspectJ compiler [1,11] to compile and weave all four aspects (which do not interfere with each other), we can use the existing Java test-generation tools, such as Parasoft Jtest 4.5 [18], to generate unit tests for the woven class. Each unit test consists of sequences of method invocations. By default, Jtest 4.5 does not generate tests that contain invocations of a public class's package-private methods; therefore, tests generated by Jtest for `Stack` do not directly invoke `iterator`.

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

```
Test 1 (T1):          Test 2 (T2):          Test 3 (T3):
 Stack s1 =            Stack s2 =            Stack s3 =
  new Stack();          new Stack();          new Stack();
 s1.push(3);           s2.push(3);           s3.push(3);
 s1.push(2);           s2.push(5);           s3.push(2);
 s1.pop();                                   s3.pop();
 s1.push(5);                                 s3.pop();
```

Raspect determines redundant tests for advised methods, advice, and intertype methods. We next discuss briefly how Raspect does this and what results it produces for the example test suite.

To determine redundant tests for *advised methods*, Raspect dynamically monitors test executions. 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 (the receiver object and method arguments) at the beginning of the execution. We call this state *method-entry state*, and its part that is related to the receiver *object state*. We represent an object using the values of the fields of all reachable objects. Two states are equivalent if their representations are the same. For instance, T2 has three method executions: a constructor without arguments is invoked, `push` adds `3` to the empty stack, and `push` adds `5` to the previous stack. We call two method executions equivalent if they are invocations of the same method on equivalent states. Raspect detects *redundant* tests for advised methods: 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 4.1). For example, Raspect detects that Test 2 is a redundant test for advised methods with respect to Test 1 because any of Test 2's three method executions is equivalent to one of Test 1's method executions. However, Test 3 is not redundant for advised methods because the last method execution `s3.pop()` is not equivalent to any of the method executions of Test 1 or Test 2.

To determine redundant tests for *advice*, Raspect dynamically monitors the execution of advice. Each test execution produces a sequence of advice executions. Similar to the definition of a method execution, each *advice execution* is characterized by the advice that is invoked and a *representation* of the state (the aspect receiver object and method arguments) at the beginning of the execution. We call this state *advice-entry state*. Raspect detects *redundant* tests for advice: a test is redundant for a test suite if every advice execution of the test is equivalent to some advice execution of some test from the suite. Because the `NonNegative` and `NonNegativeArg` aspects do not declare any fields, the advice execution is solely characterized by the advice's arguments: the target `Stack` object and the arguments of invoked methods on Stack for advice in these two aspects, respectively. Because the `Instrumentation` aspect declares only a static `count` field, which is reachable from its aspect objects, but its advice does not have any argument, the advice execution is solely characterized by the aspect object state. Raspect can detect both Test 2 and Test 3 are redundant for advice in any of the first three aspects because any advice execution of Test 2 or Test 3 is equivalent to one of the advice executions of Test 1.

To determine redundant tests for *intertype declarations* in aspects, which publicly declare methods for classes, Raspect dynamically monitors the execution of the corresponding intertype methods. Each test execution produces a sequence of intertype method executions. Similar to the definition of a method execution, each *intertype method execution* is characterized by the actual intertype method that is invoked and a *representation* of the state (method arguments)[3] at the beginning of the execution. We call this state *intertype-entry state*. Raspect detects *redundant* tests for intertype methods: a test is redundant for a test suite if every intertype method execution of the test is equivalent to some intertype method execution of some test from the suite. The `PushCount` aspect declares a `count` intertype field for the `Stack` class, which is reachable from the `Stack` object, but its `increaseCount` intertype method does not have any argument. So the intertype method execution is solely characterized by the `Stack` object state as well as the state of the `count` field declared in the aspect for the `Stack` class. Raspect can detect Test 3 is redundant for the `increaseCount` intertype method because any intertype method execution of Test 3 is equivalent to one of the intertype method executions of Test 1. However, Test 2 is not redundant for `increaseCount` because the intertype method execution generated by `s2.push(5)` is not

[2] We declare this intertype method as public for the illustration purpose so that a client can invoke the `increaseCount` method to increase `count` without invoking `push`.

[3] There is no receiver object for the intertype method because the intertype method in the aspect class is static.

equivalent to the intertype method execution generated by `s1.push(5)`. The values of the `count` field are different at the entries of these two intertype method executions.

4 Redundant-Test Detection for AspectJ

We consider three kinds of units in AspectJ programs: (1) advised methods in woven classes, (2) advice in aspect classes, and (3) intertype methods in aspect classes. We first introduce a common definition of redundant tests for all these units. Our definition is parameterized with respect to the state at the beginning of unit executions within the tests. We then describe how to minimize a test suite based on the states. We finally instantiate the definition for each of the three kinds, describing how to determine the appropriate states for advised methods, advice, and intertype methods.

To detect redundant tests for advised methods, advice, and intertype methods, we have developed Raspect, an extension of our previous Rostra framework [22] that works for Java methods and can be viewed as detecting redundant tests only for advised methods. The inputs to Raspect are a class under test and a set of methods under test; Raspect can treat an aspect class as the class under test, and pieces of advice and intertype methods as methods under test. Specifically, given an AspectJ program, Raspect performs the following steps:

1. Compile and weave aspects and base classes into class bytecode using the AspectJ compiler.
2. Generate unit tests for woven classes using the existing test-generation tools based on class bytecode, e.g., Parasoft Jtest 4.5 [18]. (For some special types of advice, we need to use our previously proposed Aspectra framework [23] to generate wrappers for the woven classes to leverage the existing test-generation tools.)
3. Compile and weave aspects and generated test classes into class bytecode using the AspectJ compiler. (This is necessary because the tests contain call sites to the base-class methods, and some aspects may be woven into each such call site, e.g., the `NonNegative` and `Instrumentation` aspects for the `Stack` class.)
4. Detect and remove redundant tests for the three kinds of units:
   - for each advised method, treat the woven class as the class under test and the advised method itself as the method under test;
   - for each piece of advice, treat the aspect class as the class under test and the advice as the method under test;
   - for each intertype method, treat the aspect class as the class under test and the intertype method as the method under test.

The use of Raspect for detecting redundant tests for advised methods, advice, and intertype methods assumes that these methods are deterministic: for each method, any two executions that begin with the same state (reachable from the receiver and method arguments) have the same behavior. In particular, this means that Raspect might not work on multi-threaded code or on code that depends on timing. However, it is still useful for developers to run tests on non-deterministic methods with Raspect as it can detect non-determinism that results in different states. Namely, Raspect collects the states reachable from the receiver and method arguments both before and after a method execution (in addition to the return values of the method execution). If Raspect detects that two executions that begin with the same state produce different states or return values, non-deterministic behavior is exposed and both executions are selected for inspection.

4.1 General Detection of Redundant Tests

Each execution of a test produces a sequence of method calls on the objects of the class under test (either the woven class or the aspect class). Each method call produces a method execution whose behavior depends on the state of the receiver object and method arguments at the beginning of the execution. We represent each method execution with the actual method that was executed and a representation of the state (reachable from the receiver object and method arguments) at the beginning of the execution. We call such a state *method-entry state*.

Raspect represents a method-entry state using the WholeState technique from our previous Rostra framework [22]. Each test focuses on the state of several objects, including the receiver object and method arguments. Locally, the state of an object consists of the values of the object's fields, but some of the fields may point to other objects, and thus, globally the state of an object consists of the state of all reachable objects. To represent the state of specific objects, Raspect traverses and collects the values of (some) fields reachable from these objects. Next section presents which fields Raspect collects for each of the three kinds of units.

During the traversal, Raspect performs a *linearization* [22] on the collected field values of reference type. Our linearization is similar to the standard Java serialization [20]: it translates an object graph into a sequence of integers. Whereas the serialization is in general under the control of the programmer and may produce arbitrary sequences, the linearization produces sequences that represent object graphs uniquely up to isomorphism. Details of how the linearization works are available elsewhere [22].

The linearization reduces the comparison of the method-entry states to the comparison of sequences of integers.

We denote with `linearize`(s) the state representation of a method-entry state s.

Definition 1 *Two method-entry states s_1 and s_2 are* equivalent *iff* `linearize`(s_1) =`linearize`(s_2).

We define equivalent method executions based on equivalent method-entry states.

Definition 2 *A* method execution $\langle m, s\rangle$ *is a pair of a method m and a method-entry state s.*

Definition 3 *Method executions $\langle m, s\rangle$ and $\langle m', s'\rangle$ are* equivalent *iff $m = m'$ and s and s' are equivalent.*

Each test execution produces several method executions. Under the assumption that equivalent method executions exhibit the same behavior, testing a method execution equivalent to a previously tested method execution provides no value in terms of increasing fault detection (for faults that do not depend on sequences) or increasing code coverage for the method.

Definition 4 *A test t is* redundant *with respect to a test suite S iff for each method execution produced by t, there exists an equivalent method execution of some test from S.*

Definition 5 *A test suite S is* minimal *iff there is no $t \in S$ such that t 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. Given a test suite S', there can be several possible test suites $S \subseteq S'$ that minimize S'. We use a simple algorithm to approximately find (near-)minimal test suites: our tool accepts a JUnit test suite, uses the JUnit framework [12] to execute the suite in the default order, and filters out the tests that are redundant with respect to the previously executed tests. Regardless of the order in which the tests from S' are executed, the total number of non-equivalent method executions (or object states) is the same. Since the inspection effort for automatically generated tests should focus on the non-equivalent method executions, it is practical for our tool to reduce the test suite based on the default order of JUnit test executions instead of searching for a minimal test suite.

4.2 Collecting States

The previous definitions use a method-entry state of the unit under test. For advised methods, the state is simply (a part of) the object graph reachable from the receiver object and arguments. We next show how to build the state for the other two kinds of units, advice and intertype methods. For both of them, the aspect class is the class under test.

4.2.1 Collecting State for Advice

We first discuss the specifics of methods that represent pieces of advice and then present the special treatment of the `JoinPoint` arguments for advice.

The receiver object of advice is an aspect object (obtained by calling `aspectOf`). There is a special treatment for inlined `around` advice in the base class. Such advice has no receiver object; the advice is a static method whose first argument is the receiver object of the advised method as presented in Section 2. The method-entry state for a method that implements a piece of advice is called the advice-entry state. We represent advice-entry states as the appropriate method-entry states (defined in Section 4.1).

The body of a piece of advice can use special variables `thisJoinPoint`, `thisJoinPointStaticPart`, and `thisEnclosingJoinPointStaticPart` to discover both static and dynamic information about the current join point [1, 11]. For example, the `NonNegativeArg` aspect shown in Figure 3 invokes `thisJoinPoint.getArgs()` to retrieve the arguments of the current join point and invokes `thisJoinPoint.getSignature().toShortString()` to get the method signature name associated with the current join point.

The AspectJ compiler first detects which special variables the body of the advice uses and then extends the signature of the advice with the corresponding arguments for these special variables. In this example, the AspectJ compiler extends the signature of the advice with one additional argument, `JoinPoint thisJoinPoint`. The `JoinPoint` type, the return type of `thisJoinPoint.getSignature()`, and other *AspectJ-library types* are in the packages whose names start with `org.aspectj`. We refer to an object of an AspectJ-library class as an *AspectJ-library object*.

At runtime, Raspect needs to carefully collect the state to avoid the information not desired in the advice-entry state. For instance, Raspect should not traverse and collect all the fields reachable from the `thisJoinPoint` argument that contains the reflective information about the current join point. In fact, only the return values of the methods transitively invoked on `thisJoinPoint` affect the behavior of an aspect execution. For example, only the return values of `thisJoinPoint.getArgs()` and `thisJoinPoint.getSignature().toShortString()` affect the behavior of the `NonNegativeArg` aspect.

Raspect specially treats the `JoinPoint` argument state during the object-field traversal for state representation. When the traversal encounters an AspectJ-library object, Raspect stops collecting the fields of the object. Instead, it captures the relevant parts of the `JoinPoint` state by collecting the values of all object fields reachable from the return of a method call invoked on an AspectJ-library ob-

ject; Raspect does this during the entire aspect execution and recursively avoids collecting the fields of an AspectJ-library object during the traversal of the fields from the return object. For example, `thisJoinPoint.getArgs()` returns an object array that holds the method arguments of the current join point. Raspect traverses and collects as a part of the advice-entry state the values of the fields reachable from these method arguments. In addition, `thisJoinPoint.getSignature()` returns an object of an AspectJ-library type `org.aspectj.lang.Signature`. Raspect does not traverse and collect the fields of this AspectJ-library object. When `toShortString()` is invoked on this object, it returns a `String` object (containing the short-form name of the method signature), so Raspect collects this string as a part of the advice-entry state.

4.2.2 Collecting State for Intertype Methods

In AspectJ, all intertype declarations are compiled into the intertype methods in aspect classes. The method-entry state for an intertype method is called the intertype-entry state. Since all intertype methods in the aspect class are static, there are no receiver objects for these methods. We represent intertype-entry states as the appropriate method-entry states (defined in Section 4.1).

Raspect minimizes a test suite for testing intertype methods. Our implementation of the redundant-test detection for intertype methods treats intertype methods as a special type of advice. In the rest of this paper, thus, we use *redundant-test detection for advice* to refer to redundant-test detection for both advice and intertype methods. However, in test generation for AspectJ programs, we still distinguish between intertype methods and advice. When the AspectJ compiler weaves intertype methods into the base class, these methods can become a part of the base-class interface. Therefore, the Java test-generation tools based on bytecode may directly generate method calls that exercise intertype methods, although the tools cannot directly generate method inputs that exercise advice.

5 Experimental Study

We have collected 12 AspectJ programs from a variety of sources (Section 5.1) to use as subjects to evaluate Raspect. We have implemented the Raspect techniques building on our previous Rostra framework [22]. We have also implemented Zhou et al.'s test-selection technique based on *aspect coverage (AC)* [24]; the technique selects a test if the test covers an aspect even if the same input to the aspect has been exercised by previously selected tests. We compare Zhou et al.'s technique with Raspect (Section 5.2). We have applied these techniques on the subject programs. The results show that Raspect can detect redundant tests among those selected by the AC technique, and the percentage of these redundant tests is usually lower than the percentage of redundant tests for advised methods, which is in turn is usually lower than the percentage of redundant tests for advice (Section 5.3). The results thus suggest that: (i) our new techniques perform better than the existing AC technique and (ii) more tests need to be inspected for testing advised methods than for testing advice. We finish with a discussion of Raspect (Section 5.4).

5.1 Subjects

Table 1 lists the 12 AspectJ subjects that we use in our experiments. The first four subjects (`NonNegative`, `NonNegativeArg`, `Instrumentation`, and `PushCount`) are the example aspects from Figure 3. `NullCheck` is a program used by Asberry to detect whether method calls return `null` [2]. Following Rinard et al. [19], we refer to the first five subjects as *basic aspects*. `Telecom` is an example from the AspectJ distribution [1] that simulates a community of telephone users. `BusinessRuleImpl` comprises two aspects of business rules for a banking system [14]. `StateDesignPattern` was developed by Hannemann and Kiczales [7] to illustrate aspect-oriented implementations of design patterns. `DCM` was developed by Hassoun et al. [10] to validate their proposed dynamic coupling metric (DCM) [9]. `ProdLine` uses intertype declarations and was developed by Lopez-Herrejon and Batory for product lines of graph algorithms [17]. `Bean` is an example used in the AspectJ primer from `http://aspectj.org` to enhance a class with the functionality of Java beans. `LoD` was developed by Lieberherr et al. to check the Law of Demeter [15]. It includes one checker for object form and another checker for class form; our study uses the checker for object form. The basic aspects, `DCM`, and `LoD` do not come with base classes; for these subjects, we use the `Stack` class from Figure 1 or its adapted version as the base class.

Our subjects include most of the programs used by Rinard et al. [19] in evaluating their classification system for aspect-oriented programs, the benchmarks used by Dufour et al. [6] in measuring the performance of AspectJ programs (available at `http://www.sable.mcgill.ca/benchmarks/`), and one of the aspect-oriented design pattern implementations by Hannemann and Kiczales [7] (available at `http://www.cs.ubc.ca/~jan/AODPs/`). Our subjects do not include several programs from the first two sources because these programs are concurrent (our Raspect framework works only on sequential programs) or GUI-based (GUI applications are not suitable for automated test generation with Jtest). Our subjects also do not include more design pattern implementations primarily because they use intertype declarations, which are also used by some of the other subjects.

5.2 Implementations

We have implemented the techniques of Raspect for detecting redundant tests for AspectJ programs by modifying Rostra, our previous tool for detecting redundant object-oriented unit tests for Java programs [22]. We reuse Rostra to detect redundant tests for advised methods, advice, and intertype methods. (Recall from Section 4.2.2 that we use *redundant tests for advice* to refer to both advice and intertype methods.) To apply Rostra, we first need to determine which classes and methods are under test for the three kinds of units (Section 4). Our implementation dynamically determines this. During class loading time, our tool checks whether a class is an aspect class by inspecting the names of its methods, base on the special names that the AspectJ compiler gives to advice. Our tool similarly detects inlined `around` advice in the base class based on their names. When detecting redundant tests for advice, the tool treats the identified aspect classes as the classes under test and the identified advice as the methods under test.

We have also implemented the AC technique [24] that has a similar goal as Raspect to reduce a test suite for aspect-oriented programs. The AC technique selects a test from the test suite if the test covers at least one piece of advice (even if the input to the advice has been exercised before). We quantitatively compare the AC technique with our proposed Raspect techniques.

5.3 Results

We first feed the woven class bytecode for each subject to Jtest 4.5 [18] to generate tests. Jtest allows the user to set the length of method sequences between one and three; we set it to three. Table 1 shows the number of tests generated by Jtest and the coverage of the branches within the aspect classes achieved by these tests. (We have adapted Hansel [8] to measure the branch coverage of aspect classes at the bytecode level.)

We then run our tools for the Raspect and AC techniques on the Jtest-generated test suites. Table 1 shows the percentage of redundant tests, the number of non-equivalent (method or advice) executions, and the number of non-equivalent (class or aspect object) states that the tools detect. The results for the AC technique are in the columns labeled "AC". (The percentage of tests selected by AC corresponds to the percentage of non-redundant tests in our context.) The results for the Raspect techniques are in the columns labeled "AM" and "AD", for advised methods and (pieces of) advice, respectively.

5.4 Discussion

We first discuss the branch coverage of the test suites. We then present more details of the redundant tests that various techniques detect. We finally consider the quality of Raspect and its relationship with test generation.

We measured the aspect branch coverage achieved by test suites minimized with different techniques. We found that the coverage of minimized suites remains the same as the coverage of the original suites, which shows that the Raspect techniques based on equivalent states preserve structural coverage for our set of subjects.

Our technique for advised methods detects the same number of redundant tests for the first three subjects in Table 1, even though it detects different numbers of non-equivalent method executions. Receall that we weave `Stack` with these aspects. The numbers differ because (i) the advice in the `NonNegative` aspect invokes `iterator()`, increasing the total number of non-equivalent method executions and (ii) for `NonNegativeArg`, the AspectJ compiler inserts an extra static initializer into the `Stack` class to enable the join-point reflection. In general, when a base class is woven with different aspects, running the same test suite on the woven class can produce different numbers of redundant tests, non-equivalent method executions, or non-equivalent object states for the advised methods.

The `PushCount` aspect has lower percentage of redundant tests, more non-equivalent method executions, and more non-equivalent object states than the first three aspects. The reason is that `PushCount` declares an intertype field and an intertype method for `Stack`, which results in more fields in states.

The `NullCheck` subject uses `around` advice for the methods with non-void returns. To provide a base class for `NullCheck`, we adapt `Stack` from Figure 1 by changing the `int` type to `Integer` and use `NullCheck` to advise both `pop` and `iterator` methods. The aspect declares no object field for itself and the inlined `around` advice is static; therefore, our technique for advice detects no aspect object state.

For the first five, basic aspects, our techniques detect more redundant tests for advice than for advised methods. For three other subjects (`Telecom`, `BusinessRuleImpl`, and `Bean`), our techniques also detect more redundant tests for advice than for advised methods. But for `StateDesignPattern`, interestingly the aspect-object states are more complicated than the base-class-object states; this phenomenon is not common among AspectJ programs. Subsequently our techniques detect fewer redundant tests for advice than for advised methods in `StateDesignPattern`. Like the states of the `StateDesignPattern` aspect, the states of the `DCM` aspect are also complicated. Interestingly our technique for advice detects no redundant tests but our technique for advised methods detects 72.7 percent of redundant tests. Because static fields defined in the `DCM` aspect store the information

AspectJ program	number of tests	branch coverage	redundant tests [%]			non-eq. executions			non-eq. states	
			AC	AM	AD	AC	AM	AD	AM	AD
NonNegative	44	6/7	6.8	72.7	90.9	71	16	5	6	1
NonNegativeArg	44	8/9	6.8	72.7	90.9	71	13	5	6	1
Instrumentation	44	4/4	0.0	72.7	84.1	106	12	8	6	4
PushCount	94	8/8	0.0	70.2	77.7	267	28	22	13	0
NullCheck	45	4/8	60.0	71.1	91.1	20	14	5	6	0
Telecom	798	19/28	0.0	95.2	98.5	5780	52	21	21	2
BusinessRuleImpl	439	14/21	50.1	94.1	97.7	268	35	12	6	2
StateDesignPattern	129	15/17	0.0	48.8	36.4	348	82	172	47	74
DCM	44	57/91	0.0	72.7	0.0	942	13	271	6	126
ProdLine	474	80/294	0.0	86.5	86.5	18057	68	206	30	6
Bean	1895	18/19	0.0	69.9	73.7	6864	1144	746	417	0
LoD	44	8/133	18.2	68.2	18.2	60	16	55	7	2

Table 1. Results of applying redundant-test detection on Jtest-generated tests using the aspect-coverage technique (AC) and the Raspect technique on advised methods (AM) and advice (AD)

of method call history, the technique for advice would still detect no redundant tests even if running the same test multiple times within a test suite.

The base classes of the ProdLine subject are a set of empty classes. Our testing focuses on one of these classes: Vertex. The woven class contains 10 intertype fields that are declared by seven aspects. It also contains four methods that are declared by two aspects: DFS and Undirected, which are developed for depth-first search and undirected graph, respectively. Because the methods under test are the same for both advice and advised methods, our techniques for advice and advised methods detect the same redundant tests for Vertex.

The LoD subject defines a Check aspect. The method arguments of advice in Check can reach the instances of Percflow and Pertarget, which bring in complex calling context information. Therefore, our techniques detect fewer redundant tests for advice than for advised methods.

We next compare the AC technique and the Raspect techniques. AC detects no redundant tests for seven subjects. For the remaining five subjects, AC detects some tests as redundant since they never cover any advice. In general, Raspect can detect more redundant tests than AC.

As we described earlier, we also compare redundant tests for advised methods (AM) and advice (AD). Our technique for advised methods can often detect a high percentage of redundant tests among those generated by Jtest. Jtest generates a relatively small number of method arguments for the methods of the class under test and generates many different combinations of method sequences with these arguments. Thus, Jtest can produce a lot of redundant tests for Java programs, as observed in the Rostra experiments [22], which also showed that removing these redundant tests does not decrease the fault detection capability and structural coverage achieved by the test suite. Our Raspect experiments show that Jtest also generates a lot of redundant tests for advised methods. Moreover, our technique for advice usually detects an even higher percentage of redundant tests, which means that fewer automatically generated tests need to be manually inspected when focusing on advice than when focusing on advised methods.

Our tool outputs traces of state information for non-equivalent (method or advice) executions and (class or aspect) object states; the user can inspect these traces for correctness. Aspect classes typically contain fewer object fields than the base classes; therefore, the size of the exercised state space of aspect objects is smaller than the size of the exercised state space of objects of base classes. Three interesting exceptions are StateDesignPattern, DCM, and LoD; for these subjects, Raspect detect fewer redundant tests for advice than for advised method.

Our evaluation uses Jtest for test generation. While the specific numbers (Table 1) depend on the generated test suites (as well as the subjects, how they use aspects, etc.), Raspect is not specific to Jtest. Raspect may work with any other test-generation tool, and the percentage of redundant tests may increase or decrease.

We expect that Raspect can be applied to a wide range of AspectJ programs. The results show that Raspect can substantially reduce the size of the (Jtest-)generated test suites for manual inspection when specifications are absent, a common case in practice. We expect that test inspection for advice would require less effort than test inspection for advised methods for most AspectJ programs. Our results also show some cases in which a larger inspection effort is needed for advice than for advised methods. Raspect is still useful for such programs as it can improve the developer's understanding of the aspect behavior by drawing the attention to the dominating behavior.

Raspect is primarily based on a dynamic analysis; to de-

tect redundant tests for advice or advised methods, Raspect needs to run the generated tests. Based on the results with Jtest and the 12 subjects, Jtest's test generation is relatively expensive for large programs, but the execution of the generated tests is usually cheap with a reasonable runtime overhead incurred by our dynamic analysis.

6 Conclusion

We have proposed Raspect, a framework for detecting redundant unit tests for AspectJ programs. Redundant tests are defined for three kinds of units: advised methods, pieces of advice, and intertype methods. We have formally defined inputs to these units based on object states. Raspect extends our previous Rostra framework for detecting redundant tests for (advised) methods with detection of redundant tests for advice and intertype methods. Our focus is on detecting and removing redundant tests before the manual inspection of automatically generated tests. Raspect allows us to generate tests for AspectJ programs by reusing the existing Java test-generation tools and postprocessing the generated test suites.

Acknowledgments

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. We thank Alex Salcianu for the valuable feedback on an earlier version of this paper.

References

[1] AspectJ compiler 1.2, May 2004. `http://eclipse.org/aspectj/`.

[2] R. D. Asberry. Aspect oriented programming (AOP): Using AspectJ to implement and enforce coding standards. Draft manuscript, 2002.

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

[4] L. Bergmans and M. Aksits. Composing crosscutting concerns using composition filters. *Commun. ACM*, 44(10):51–57, 2001.

[5] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34:1025–1050, 2004.

[6] B. Dufour, C. Goard, L. Hendren, O. de Moor, G. Sittampalam, and C. Verbrugge. Measuring the dynamic behaviour of AspectJ programs. In *Proc. 19th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 150–169, 2004.

[7] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proc. 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 161–173, 2002.

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

[9] Y. Hassoun, R. Johnson, and S. Counsell. A dynamic runtime coupling metric for meta-level architectures. In *Proc. 8th European Conference on Software Maintenance and Reengineering*, pages 339–346, 2004.

[10] Y. Hassoun, R. Johnson, and S. Counsell. Emprical validation of a dynamic coupling metric. Technical Report BBKCS-04-03, Birbeck College London, March 2004.

[11] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In *Proc. 3rd International Conference on Aspect-Oriented Software Development*, pages 26–35, 2004.

[12] JUnit, 2003. `http://www.junit.org`.

[13] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. 11th European Conference on Object-Oriented Programming*, pages 220–242. 1997.

[14] R. Laddad. *AspectJ in Action*. Manning, 2003.

[15] K. Lieberherr, D. H. Lorenz, and P. Wu. A case for statically executable advice: checking the law of demeter with aspectj. In *Proc. 2nd International Conference on Aspect-Oriented Software Development*, pages 40–49, 2003.

[16] K. Lieberherr, D. Orleans, and J. Ovlinger. Aspect-oriented programming with adaptive methods. *Commun. ACM*, 44(10):39–41, 2001.

[17] R. E. Lopez-Herrejon and D. Batory. Using AspectJ to implement product-lines: A case study. Technical report, University of Texis at Austin, September 2002.

[18] Parasoft. Jtest manuals version 4.5. Online manual, April 2003. `http://www.parasoft.com/`.

[19] M. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. In *Proc. 12th International Symposium on the Foundations of Software Engineering*, pages 147–158, 2004.

[20] Sun Microsystems. Java 2 Platform, Standard Edition, v 1.4.2, API Specification. Online documentation, Nov. 2003. `http://java.sun.com/j2se/1.4.2/docs/api/`.

[21] P. Tarr, H. Ossher, W. Harrison, and J. Stanley M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *Proc. 21st International Conference on Software Engineering*, pages 107–119, 1999.

[22] T. Xie, D. Marinov, and D. Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *Proc. 19th IEEE International Conference on Automated Software Engineering*, pages 196–205, Sept. 2004.

[23] T. Xie and J. Zhao. A framework and tool supports for generating test inputs of aspectj programs. In *Proc. 5th International Conference on Aspect-Oriented Software Development*, pages 190–201, March 2006.

[24] Y. Zhou, D. Richardson, and H. Ziv. Towards a practical approach to test aspect-oriented software. In *Proc. 2004 Workshop on Testing Component-based Systems, Net.ObjectiveDays*, Sept. 2004.