Detecting Redundant Unit Tests for AspectJ Programs

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

[1] Department of Computer Science & Engineering, University of Washington, USA
[2] Department of Computer Science & Engineering, Fukuoka Institute of Technology, Japan
[3] MIT Computer Science and Artificial Intelligence Laboratory, Cambridge, MA 02139, USA
{taoxie,notkin}@cs.washington.edu, zhao@cs.fit.ac.jp, marinov@lcs.mit.edu

Technical Report UW-CSE-04-10-03
October 2004

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 tests, and it is time-consuming to manually inspect them all. This paper proposes an automated approach for detecting redundant unit tests for AspectJ programs. We introduce two levels of units in testing AspectJ programs–the higher level of the advised methods and the lower level of the pieces of advice–and 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 minimized set of tests. We have implemented our approach and our experience has shown that it 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 [5, 10, 14, 19]. 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 [4]. Automated software testing, and in particular test generation, can significantly reduce this cost. Although AOSD can lead to better-quality software, AOSD does not provide the correctness by itself. 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 shield against mistakes made by programmers or a lack of understanding of the specification. As a result, software testing remains an important task in AOSD.

Aspect-oriented programming languages, such as AspectJ [10], 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.

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 [27]. Unit testing is gaining importance with the wider adoption of Extreme Programming [3]. For aspect-oriented programs, the basic testing unit can be either an aspect or a class. In unit testing, developers isolate the unit to run independently from its environment. This allows writing small testing code that exercises 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. This also allows for testing the complex interactions between the aspect and the affected classes.

We can use 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 [16]) or as research prototypes (e.g., JCrasher [6] and Eclat [15]). These tools test a class by generating and executing numerous method sequences on the objects of the class. Since typical programs do not have executable specifications for automatic correctness checking, these tools rely on the programmer to inspect the executions of the generated tests for correctness.

Our previous work [21] has shown that automatic test-generation tools may generate a large number of *redundant* tests that do not exercise new behaviors of the class under test. Such tests only increase 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.) A key issue in automated testing is to avoid such redundant tests. This not only reduces the test generation and execution time, but also reduces the time that developers need to spend inspecting the tests.

We propose in this paper an automated approach for detecting redundant unit tests for AspectJ programs. We consider two levels of units in AspectJ programs: the pieces of advice and the advised methods. We formalize the inputs to these units using object states. Our approach starts from the unit tests that the existing tools for Java generate for the aspect and the class affected by it. Our approach then detects those tests that do not exercise new inputs to a unit (either a piece of advice or an advised method). These tests are redundant and can be removed from the generated test suite without decreasing its quality. Programmers can then inspect this much smaller set of non-redundant tests.

This paper makes the following main contributions:

- We propose an approach for detecting redundant unit tests for aspect-oriented programs; to the best of our knowledge, this is the first such approach.
- We presents an implementation of the approach for detecting redundant unit tests for AspectJ programs.
- We describe our experience in applying the approach to various AspectJ programs. The experience shows that our approach can effectively reduce the size of generated test suites for inspecting AspectJ program behavior.

2 AspectJ

We next present some details of AspectJ [1], a widely used aspect-oriented programming language. Our implementation for detecting redundant tests operates on AspectJ programs, but the underlying ideas apply to the general class of aspect-oriented languages.

AspectJ is a seamless, aspect-oriented extension to Java. AspectJ adds to Java several new concepts and associated constructs, including join points, pointcuts, advice, inter-type declarations, and aspects. An *aspect* is a modular unit of crosscutting implementation in AspectJ. Each aspect encapsulates functionality that crosscuts other 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 reaches a pointcut. Advice is a method-like mechanism which consists of instructions that execute *before*, *after*, or *around* a pointcut. The around advice executes *in place* of the indicated pointcut, which allows the aspect to replace a method.

The AspectJ compiler ensures that the base and aspect code run together in a properly coordinated fashion. The compiler does this using *aspect weaving* which composes the code of the base class and the aspect to ensure that applicable advice runs at the appropriate join points [1, 8]. Details of AspectJ are available elsewhere [1].

3 Example

We next illustrate how our approach determines redundant tests for an AspectJ program. We use a simple integer stack example adapted from Rinard *et al.* [17]. Figure 1 shows the implementation of the class. This public class has two public non-constructor methods: `push` and `pop` are standard stack operations, and one package-private method: `iterator` returns an iterator that can be used to traverse the items in the stack. Figure 2 shows three aspects for the stack class: `NonNegative`, `NonNegativeArg`, and `Instrumentation`.

The 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 integers. The advice is executed before any call to a `Stack` method. The `NonNegativeArg` aspect checks whether `Stack` method arguments are nonnegative integers. The aspect contains a piece of advice that goes through all arguments of an about to be executed `Stack` method to check whether they are nonnegative integers. The advice is executed before the execution of any `Stack` method. The `Instrumentation` aspect counts the number of times a `Stack`'s `push` method is invoked on an object since its creation[1]. The aspect contains a piece of advice that increments the static `count` field defined in the aspect. The advice is executed after any call to `Stack`'s `push` method. The aspect

[1] The advice implementation works correctly only when no interleaving among stack objects' `push` and constructor calls.

```
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); }
}
```

Fig. 1. An integer stack implementation

contains another piece of advice that resets the static `count` field. This piece of advice is executed after any call to `Stack`'s constructor.

We use the AspectJ compiler [1,8] to compile and weave either of three aspects and the stack class (called *base class*) into class bytecode (called *woven class*); a method in a woven class is called an *advised method*. An aspect is compiled to an *aspect class* and a piece of advice in the aspect is compiled into an *advice method* in the aspect class. After the compilation, we can use existing Java test-generation tools, such as Parasoft Jtest 4.5 [16], 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 = new Stack();     Stack s2 = new Stack();     Stack s3 = 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();
```

To determine redundant tests for advised methods, our techniques dynamically monitor 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 (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, `push` adds `5` to the previous stack. We call two method executions equivalent if they are invocations of the same method on equivalent states. Our techniques detect *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, our techniques detect that Test 2 is a

```
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++;    }
}
```

Fig. 2. `NonNegative`, `NonNegativeArg`, and `Instrumentation` aspects

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, our techniques dynamically monitor the execution of advice methods. Each test execution produces a sequence of advice executions. Similar to the definition of a method execution, each *advice execution* is characterized by the actual advice method that is invoked and a *representation* of the state (aspect receiver object and method arguments) at the beginning of the execution. Our techniques detect *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 (Section 4.2). Because the `NonNegative` and `NonNegativeArg` aspects do not declare any fields, the advice execution is solely characterized by the advice method 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 method does not have any argument, the advice execution is solely characterized by the aspect object state. Our techniques can detect both Test 2 and Test 3 are redundant for advice in any of the three aspects because any advice execution of Test 2 or 3 is equivalent to one of the advice executions of Test 1.

4 Redundant-Test Detection for AspectJ Programs

The AspectJ compiler [1, 8] compiles each aspect into a standard Java class (called *aspect class*) and each piece of advice from the aspect into a standard Java method (called *advice method*) of the aspect class. The compiler also inserts the calls to these advice methods at the appropriate locations in the base classes. These base classes are then called *woven classes* and the methods in these classes are called *advised methods*.

We define redundant tests for AspectJ programs at two levels: (1) redundant tests for advised methods in woven classes and (2) redundant tests for advice methods in aspect classes. We use our existing Rostra framework [21] to detect redundant tests for advised methods. We also extend Rostra to detect redundant tests for advice methods. Rostra takes as inputs the class and methods under test, so we treat an aspect class as the class under test and an advice method as the method under test. Specifically, given an AspectJ program, our extended Rostra 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 existing test generation tools that are based on class bytecode (e.g., Parasoft Jtest 4.5 [16]).
3. Compile and weave aspects and generated test classes into class bytecode using the AspectJ compiler. (This is necessary because some aspects may be woven into each call site to the base-class methods, e.g., the `NonNegative` and `Instrumentation` aspects for the `Stack` class, and the tests contain such call sites.)
4. Detect and remove redundant tests for advised methods using the Rostra framework [21] (Section 4.1).
5. Detect and remove redundant tests for advice methods by treating an aspect class as the class under test and an advice method as the method under test (Section 4.2).

The use of Rostra for detecting redundant tests for advised and advice 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 will behave the same. In particular, this means that Rostra might not work on multi-threaded code or on code that depends on timing.

Our approach detects redundant tests based on the state at the beginning of method executions within the tests. We next describe how the state is determined for advised and advice methods. We then describe how we minimize a test suite based on these states.

4.1 Detecting Redundant Tests for Advised Methods

Each execution of a test produces a sequence of method calls on the objects of the woven class (under test). 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. Each method execution can be represented with the actual method that is invoked and a representation of the state (reachable form receiver object and method arguments) at the beginning of the execution. We call such a state *method-entry state*.

We represent a method-entry state in this work using the WholeState technique from the Rostra framework [21]. Each tests 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, we traverse and collect the values of fields reachable from these objects. During the traversal, we perform a linearization algorithm [21] on the collected field values with reference types. In particular, when we encounter a reference-type field `f`, instead of collecting its value (which is a memory address) in the state representation, we collect the following representation:

- collect "null" if (`f` == `null`).
- collect "not_null" if (`f` != `null`) and there exists no previously encountered field `f′` such that (`f` == `f′`).
- collect `fname′` where `fname′` is the name of the earliest encountered field `f′` such that (`f` == `f′`) and (`f` != `null`).

The linearization does not traverse the fields of basic objects–e.g., a `String` or a primitive-wrapper object such as `Integer`–whose `toString()` methods return a unique string. Instead, the linearization uses this string in the state representation.

The state representation of a method-entry state is basically a sequence of strings. Comparing two state representations is reduced to comparing two sequences of strings. 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).

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

Definition 3. *Two method executions $\langle m_1, s_1\rangle$ and $\langle m_2, s_2\rangle$ are* equivalent *iff*

(1) $m_1 = m_2$, and
(2) `linearize`(s_1) =`linearize`(s_2).

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 one does not provide any new value in terms of increasing fault detection or code coverage for the method.

Definition 4. *A test t is* redundant *in testing advised methods for 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. Figure 3 shows the pseudo-code of the test minimization algorithm. The algorithm receives an original test suite and produces a minimal test suite. It runs each test in the original test suite and collect the method executions produced by the execution of the test. If all the method executions produced by the test are a subset (in terms of equivalence) of the existing method executions that are produced by previously executed tests, the test is a redundant test

```
Set minimization(Set origTests) {
  Set methodExecs = new Set();
  Set nonRedundantTests = new Set();
  foreach (Test t in origTests) {
     Set curMethodExecs = runAndCollect(t);
     if !(curMethodExecs subset methodExecs) {
         nonRedundantTests.add(t);
         methodExecs = methodExecs union curMethodExecs;
     }
  }
  return nonRedundantTests;
}
```

Fig. 3. Pseudo-code of the test minimization algorithm.

and is discarded; otherwise, the test is a non-redundant test and its produced method executions are added to the existing method executions. Given a test suite S', there can be several possible test suites $S \subseteq S'$ that minimize S', depending on the order of running the tests in S'. In our implementation, our algorithm accepts a JUnit test suite and uses the test-execution order enforced by the JUnit framework [9].

4.2 Detecting Redundant Tests for Advice

This section shows how we treat aspect class as the class under test and advice methods as public methods in the class under tests, and then adapt the approach presented in Section 4.1 for detecting redundant tests for advice methods. We also discuss how the AspectJ compiler compiles pieces of advice into advice methods, and present the definitions of redundant tests for advice methods. We finally illustrate the special treatments of JoinPoint arguments for advice methods.

Compilation of Advice The AspectJ compiler compiles each aspect into an aspect class and each piece of advice in the aspect into a public non-static method (called *advice method*) in the aspect class [1, 8]. The parameters of this public method are the same as the parameters of the advice, possibly in addition to some `thisJoinPoint` parameters (discussed separately in this section). 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 compiled advice methods. At each site of these inserted calls, a singleton object of an aspect class is first obtained by calling the static method `aspectOf`, which is defined in the aspect class. Then an advice method is invoked on the aspect object.

Both pieces of before and after advice are compiled into advice 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. But it takes one additional argument: an `AroundClosure` object. A call to `proceed` in the around advice method 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 methods of the base class; no around advice method is created in the aspect class for this case.

Redundant Tests for Advice Methods Except for the type of inlined around advice, all pieces of advice are compiled into advice methods in aspect classes. In unit testing of an AspectJ program, we treat aspect classes and their advice methods as lower-level units comparing to the woven classes and their public methods.

Similar to the way of defining method-entry state, we can define advice-entry state for an advice method. An *advice-entry state* is the representation of the state (receiver object and method arguments) at the beginning of the advice's execution. The receiver object of an advice method is an aspect object (obtained by calling `aspectOf`). We also treat advised methods in the base class where around advice is inlined as special advice methods. The receiver objects of these special methods are the object of the base class. We represent advice-entry states similar to representing the method-entry states (discussed in Section 4.1). Next we define advice-entry state equivalence, advice execution, their equivalence, and redundant tests for advice methods, similar to the corresponding ones for advised methods.

Definition 6. *Two advice-entry states s_1 and s_2 are equivalent iff*

$$\texttt{linearize}(s_1) = \texttt{linearize}(s_2).$$

Definition 7. *An* advice execution $\langle a, s\rangle$ *is a pair of an advice method a and an advice-entry state s.*

Definition 8. *Two advice executions $\langle a_1, s_1\rangle$ and $\langle a_2, s_2\rangle$ are* equivalent *iff*

(1) $a_1 = a_2$, and
(2) $\texttt{linearize}(s_1) = \texttt{linearize}(s_2)$.

Definition 9. *A test t is* redundant *in testing advice methods for a test suite S iff for each advice execution produced by t, there exists an equivalent advice execution of some test from S.*

Then we can adapt the test minimization algorithm shown in Figure 3 for minimizing a test suite for testing advice methods.

thisJoinPoint Arguments The body of a piece of advice can use three special variables, i.e., `thisJoinPointStaticPart`, `thisEnclosingJoinPointStaticPart`, and `thisJoinPoint` to discover both static and dynamic information about the current join point [1, 8]. The AspectJ compiler detects which special variables are referred within the body of the advice and extends the signature of the advice method with corresponding parameters for these referred special variables. For example, the `NonNegativeArg` aspect shown in Figure 2 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 extends the signature of the advice method with one additional parameter:

```
JoinPoint thisJoinPoint.
```

The JoinPoint type and the return type of **`thisJoinPoint`**`.getSignature()` are in the packages whose names start with "`org.aspectj.`" We refer to a type defined in these packages as an *AspectJ-library type*.

At runtime, if we are not careful in collecting the state, we may collect more information than that desired as the advice-entry state. This would happen if we traversed and collected all the fields reachable from the **`thisJoinPoint`** argument, which contains reflective information about the current join point. In fact, the aspect execution behavior is affected only by the return values of those method calls transitively invoked on **`thisJoinPoint`**. For example, only the return values of

`thisJoinPoint``.getArgs()`, and

`thisJoinPoint``.getSignature().toShortString()`

are relevant for affecting the behavior of the `NonNegativeArg` aspect.

To address this issue of JoinPoint argument state, we use a special treatment during object-field traversal for state representation. When we encounter an AspectJ-library-type object during the traversal, we stop collecting the fields of the object. Instead, we capture the relevant parts of the JointPoint state by collecting and traversing the values of all object fields reachable from the return of a method call if the method call is invoked on an AspectJ-library-type object within the aspect execution (during this return-object-field traversal, we still avoid collecting the fields of an AspectJ-library-type object). For example, the return of **`thisJoinPoint`**`.getArgs()` is an object array to hold the method arguments of the current join point. We traverse and collect the values of fields reachable from these method arguments as part of the advice-entry state. In addition, the return of **`thisJoinPoint`**`.getSignature()` is an object with an AspectJ-library type: `org.aspectj.lang.Signature`. We do not traverse and collect the object fields of this signature object because it is an AspectJ-library-type object. Then `toShortString()` is invoked on this signature object and the method return is a `String`, containing the short-form name of the method signature. We also collect this string as part of the advice-entry state.

5 Experience

We have implemented a test-minimization tool for AspectJ programs by modifying Rostra, our previous tool for detecting redundant object-oriented unit tests [21]. We have applied our tool on a variety of AspectJ programs. This section describes our experience on several typical AspectJ programs. Most of these programs were used by Rinard et al. [17] in evaluating their classification system for aspect-oriented programs. Table 1 lists the programs that we use. For each program, we first feed its woven class bytecode to Jtest 4.5 [16] to generate tests. (Jtest allows the user to set the length of sequences between one and three, and we set it as three.) The second column of Table 1 shows the number of tests generated by Jtest. After weaving the base classes, aspects, and generated tests, we run these generated tests with our tool for detecting redundant tests for advised methods. We then rerun these tests with the tool for detecting redundant tests for advice. For either run, the tool reports the percentage of redundant tests (Columns 3 and 4), the number of non-redundant tests (Columns 5 and 6), the number of nonequivalent method or advice executions (Columns 7 and 8), and the number of nonequivalent class or aspect object states (Columns 9 and 10). The results for advised methods and advice are put in the columns with the titles of “meth” and “adv”, respectively.

Basic Aspects. Many aspects (such as the ones shown in Figure 2) log or check base classes. Our tool detects the same percentage of redundant tests for advised methods on

AspectJ program	tests	%r-tests		nr-tests		ne-methexec		ne-objstates	
		meth	adv	meth	adv	meth	adv	meth	adv
NonNegative	44	72.7	90.9	12	4	16	5	6	1
NonNegativeArg	44	72.7	93.2	12	3	13	6	6	1
Instrumentation	44	72.7	84.1	12	7	12	8	6	4
Telecom	798	95.2	98.5	38	12	52	21	21	2
BusinessRuleImpl	439	94.1	97.7	26	10	35	12	6	2
StateDesignPattern	129	48.8	36.4	66	82	82	172	47	74

Table 1. Results of applying the test-minimization tool on Jtest-generated tests

the three aspects in Figure 2 (shown in the first three data rows of Table 1). However, interestingly the tool detects different numbers of nonequivalent method executions when we weave `Stack` with these aspects. The numbers are different because the advice in the `NonNegative` aspect invokes `stack.iterator()`, increasing the total number of nonequivalent method executions, and the AspectJ compiler inserts an extra static initializer method `<clinit>` into the `Stack` class for preparing the joinpoint reflection within `NonNegativeArg`. In general, when a base class is woven with different aspects, running the same test suite on the woven class with our tool might produce different numbers of non-redundant tests, nonequivalent method executions, or nonequivalent object states for advised methods.

Telecom. The Telecom program is an example available with the AspectJ distribution [1]. It simulates a community of telephone users. One key base class is `Connection`, which has six non-constructor methods. It has an integer field `state` to indicate the state of the connection. The `Timing` aspect records the phone connection time (we have replaced a `Timer` class's `startTime` and `stopTime` with some constants to make method executions deterministic for testing purposes) and the `Billing` aspect uses the connection time to bill the dialer. Neither aspect declares any object field for itself but declare some object fields for other classes. Therefore, our tool detects the states of these two aspects to be empty at runtime (only two nonequivalent aspect states are exercised during test execution). The `Timing` aspect declares two pieces of `after` advice for the `call` of `Connection`'s `complete()` and `drop()` methods and the arguments of these pieces of advice are the target `Connection` object of these method calls. Therefore, our tool determines that the inputs to these pieces of advice are the `Connection` object states. The `Billing` aspect also defines two pieces of `after` advice. One piece of advice is for the `call` of `Connection`'s constructor; the advice's argument is the constructor's first argument (indicating the dialer). The other piece of advice is for the `call` of `Connection`'s `drop()` and its argument is the target `Connection` object. Some method executions of `Connection` do not produce any advice execution and some nonequivalent method executions produce equivalent advice executions because only some parts of method inputs (receiver objects or arguments) are visible and usable to advice executions. Our tool detects that about one third of the non-redundant tests for advised methods are in fact redundant for advice.

Aspects for Business Rule Implementation. Two aspects of business rules for a banking system are used as examples in Section 12.5 of [12]. `MinimumBalanceRuleAspect` defines a piece of `before` advice for the execution of `Account`'s `debit` method. Another OverdraftProtectionRuleAspect defines another piece of before advice for the execution of Account's debit method [2]. Neither aspect declares any object field for itself; our tool detects only two nonequivalent aspect states exercised by the generated tests. The arguments to both pieces of advice are the target Account object and the argument (the withdrawal amount) of the debit. Our tool detects that about 40 percent of the non-redundant tests for advised methods are in fact redundant for advice.

State Design Pattern Aspect. The state design pattern had been implemented using AspectJ by Hannemann and Kiczales [7]. QueueStateAspect declares three object fields with types of QueueEmpty, QueueNormal, and QueueFull. The base class Queue declares a state field that can be assigned with any of these three fields. QueueStateAspect declares three pieces of after advice, one for the call of Queue's constructor (the advice assigns QueueEmpty to state), one for the call of Queue's insert (the advice assigns QueueNormal or QueueFull to state), one for the call of Queue's removeFirst (the advice assigns QueueNormal or QueueEmpty to state). Interestingly the aspect-object states are more complicated than the base-class-object states; this phenomenon is not common among AspectJ programs. Subsequently the tool detects that non-redundant tests for advice are more than the ones for advised methods.

Summary. Our tool can often detect a high percentage of redundant tests for advised methods among the tests generated by Jtest. This phenomenon has been observed in the experiments of evaluating Rostra [21]. Furthermore, our tool can detect an even higher percentage of redundant tests for advice. Thus, usually fewer tests need to be inspected when focusing on advice rather than advised methods. Our tool outputs traces of state information for nonequivalent method/advice executions and class/aspect object states; the user can inspect these traces for correctness. Usually an aspect-object state is empty or contains fewer object fields than base classes; therefore, the size of the exercised aspect-object state space is smaller than the size of the exercised base-class-object state space. One interesting exceptional case is the state design pattern aspect; for this aspect, we detect more non-redundant tests for advice than for advised method.

6 Related Work

The work in this paper is built on our previous work on Rostra, a framework for detecting redundant unit tests of object-oriented programs [21]. We use the definition of equivalent method executions (proposed in Rostra) for advised methods and extend the definition to equivalent aspect executions for advice. Two nonequivalent method executions might still produce the same set of nonequivalent aspect executions. This phenomenon is similar to the observation in Rostra: two tests with different method sequences might still produce the same set of nonequivalent method executions.

Souter *et al.* [18] identified the code associated with a particular maintenance task (referred as a concern) and performed testing tasks with respect to the concern. They

[2] The original version in the book [12] requires the debit execution for the advice to be invoked underneath a method in a CheckClearanceSystem class; we comment this constraint out for the generated tests to exercise both aspects.

instrumented only the concern for collecting runtime information so that they could reduce the space and time cost of running tests. They also suggested organizing tests according to concerns, so that tests could be selected or prioritized given a concern. A concern in an application corresponds to an aspect in an AspectJ program. Their approach selects a test if the test covers the aspect even if the same input to the aspect has been exercised by previously selected tests. Zhou *et al.* [26] used the same approach for selecting relevant tests for an aspect. When testing an aspect using our approach, we selects a test if the test covers the aspect and the input to the aspect is different from any previously exercised input. Therefore, our approach selects fewer tests with respect to an aspect than these two previous approaches but preserves the same quality of tests selected by these approaches for testing the aspect.

Xu *et al.* [22] proposed a testing approach for aspect-oriented programs based on an aspect flow graph. The graph consists of an aspect scope state model (ASSM), which is a combination of class and aspect state models and the method and advice flow graphs, and can be used to represent some classes together with some additional advice that may affect the behavior of these classes. Based on the ASSM of an aspect-oriented program, one can produce necessary test suites for testing the program. Their approach first analyzes how the behaviors of classes can be dynamical affected by aspects, and then constructs an ASSM to generate a set of code based test cases. They also define a test coverage criterion for testing aspect-oriented programs. Their work focuses on how to construct some abstract models for supporting aspect-oriented testing, whereas our work focuses mainly on how to reduce the redundant unit tests when performing unit testing on AspectJ programs using existing test-generation tools such as Jtest [16].

Alexander *et al.* [2] proposed a fault model for aspect-oriented programming, which includes six types of faults that may occur for aspect-oriented systems. Although the model is useful for guiding the development of testing coverage tools for aspect-oriented programs, unlike our approach, it does not provide a concrete method for testing aspect-oriented programs.

Zhao [23, 24] proposed a data-flow-based unit testing approach for aspect-oriented programs. For each aspect or class, the approach performs three levels of testing: intra-module, inter-module, and intra-aspect or intra-class testing. His work focused on unit testing of aspect-oriented programs based on data flow, whereas our work focuses on detecting redundant unit tests for AspectJ program based on object states, in order to reduce the unit testing cost.

Krishnamurthi *et al.* [11] verified aspect advice modularly by formally modeling a program fragment as a state machine and a piece of advice as a state machine. They treated joinpoints as function calls. Our approach also implicitly models a piece of advice as a state machine and focuses on testing aspects.

Zhao and Rinard [25] developed Pipa, a behavioral interface specification language (BISL) for AspectJ for formal verification. Pipa is a simple and practical extension to Java Modeling Language (JML) [13], a BISL designed for Java. Pipa uses a similar way as JML to specify AspectJ classes and interfaces, and extends JML, with just a few new notations, to specify AspectJ aspects. By transforming an AspectJ program together with its Pipa specification into a standard Java program with JML specification, one can formally verify AspectJ programs by using existing JML-based tools. Programmers can write specifications in Pipa for AspectJ programs and use them for correctness checking during test execution thus avoiding inspection efforts. However, when running generated tests is relatively expensive for regression testing, we can still minimize or prioritize generated tests for regression testing based on equivalent method executions or advice executions proposed in our approach.

7 Conclusion and Future Work

We have proposed an automated approach for detecting redundant unit tests for AspectJ programs. Redundant tests are defined for two types of units: advised methods and pieces of advice. We have formally defined inputs to either type of units based on object states. We have used the Rostra framework [21] to detect redundant tests for advised methods and adapted the framework to detect redundant tests for advice. In this work, we have focused on detecting redundant tests and removing them before the inspection of test executions. By doing so, we can still generate tests for AspectJ programs by reusing existing Java test-generation tools and then postprocess their generated tests by removing redundant tests for AspectJ programs.

In future work, we plan to develop techniques and tools for directly generating non-redundant tests for AspectJ programs. In testing units of advised methods, we can use the test generation techniques for Java programs by avoiding redundant tests [20]. In test generation for an advised method, we have direct controls on choosing which inputs to exercise the method. In test generation for a piece of advice, we do not have direct controls on the inputs to the advice but have control on inputs to advised methods, which invoke the advice. We have shown that different inputs to an advised method can produce the same input for a piece of advice. Before we actually run these different inputs to the advised method, we plan to use static or dynamic analysis to determine which parts of inputs to the advised method are relevant to the inputs to the advice. Then we can predict whether an input to the advised method could lead to a new input to the advice. Based on this prediction, we can generate inputs to the advised method that can lead to new inputs to the advice with a high probability.

References

1. AspectJ compiler 1.2, May 2004. `http://eclipse.org/aspectj/`.
2. R. T. Alexander, J. M. Bieman, and A. A. Andrews. Towards the systematic testing of aspect-oriented programs. Technical Report CS-4-105, Department of Computer Science, Colorado State University, Fort Collins, Colorado, 2004.
3. K. Beck. *Extreme programming explained*. Addison-Wesley, 2000.
4. B. Beizer. *Software Testing Techniques*. International Thomson Computer Press, 1990.
5. L. Bergmans and M. Aksits. Composing crosscutting concerns using composition filters. *Commun. ACM*, 44(10):51–57, 2001.
6. C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34:1025–1050, 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. E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In *Proc. 3rd International Conference on Aspect-Oriented Software Development*, pages 26–35, 2004.

9. JUnit, 2003. `http://www.junit.org`.
10. 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.
11. S. Krishnamurthi, K. Fisler, and M. Greenberg. Verifying aspect advice modularly. In *Proc. 12th International Symposium on the Foundations of Software Engineering*, November 2004.
12. R. Laddad. *AspectJ in Action*. Manning, 2003.
13. 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.
14. K. Lieberherr, D. Orleans, and J. Ovlinger. Aspect-oriented programming with adaptive methods. *Commun. ACM*, 44(10):39–41, 2001.
15. C. Pacheco and M. D. Ernst. Eclat documents. Online manual, Oct. 2004. `http://people.csail.mit.edu/people/cpacheco/eclat/`.
16. Parasoft. Jtest manuals version 4.5. Online manual, April 2003. `http://www.parasoft.com/`.
17. 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*, November 2004.
18. A. L. Souter, D. Shepherd, and L. L. Pollock. Testing with respect to concerns. In *Proc. International Conference on Software Maintenance*, page 54, 2003.
19. 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.
20. T. Xie, D. Marinov, and D. Notkin. Improving generation of object-oriented test suites by avoiding redundant tests. Technical Report UW-CSE-04-01-05, University of Washington Department of Computer Science and Engineering, Seattle, WA, Jan. 2004.
21. 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.
22. W. Xu, D. Xu, V. Goel, and K. Nygard. Aspect flow graph for testing aspect-oriented programs. In *Proc. 8th IASTED International Conference on Software Engineering and Applications*, 2004.
23. J. Zhao. Tool support for unit testing of aspect-oriented software. In *Proc. OOPSLA'2002 Workshop on Tools for Aspect-Oriented Software Development*, Nov. 2002.
24. J. Zhao. Data-flow-based unit testing of aspect-oriented programs. In *Proc. 27th IEEE International Computer Software and Applications Conference*, pages 188–197, Nov. 2003.
25. J. Zhao and M. Rinard. Pipa: A behavioral interface specification language for AspectJ. In *Proc. Fundamental Approaches to Software Engineering*, pages 150–165, April 2003.
26. 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 (TECOS 2004), Net.ObjectiveDays*, Sept. 2004.
27. H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.