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; Revised December 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 Aspectra, a framework for detecting redundant unit tests for AspectJ programs. We introduce three levels of units in testing AspectJ programs: advised methods, advice, and intertype methods, and show how to detect at each level redundant tests that do not exercise new behavior. We also apply static analysis to improve the precision of the detection. 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 Aspectra and applied it on 12 subjects taken from a variety of sources; 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 [6, 15, 20, 29]. 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 [5]. 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 protect against mistakes made by programmers. As a result, software testing remains an important task in AOSD.

Aspect-oriented programming languages, such as AspectJ [15], 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 [38]. Unit testing is gaining importance with the wider adoption of Extreme Programming [4]. 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 [23]) or as research prototypes (e.g., JCrasher [7] and Eclat [22]). 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 developers to inspect the executions of the generated tests for correctness.

Our previous work [31] proposed the Rostra framework to show that automatic test-generation tools may generate a large number of **redundant** tests that do not exercise new behavior 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.

In this paper, we propose Aspectra, a novel framework for detecting redundant unit tests for AspectJ programs. Aspectra has extended our previous Rostra framework [31], which detects redundant tests for Java methods, in two important ways. First, Aspectra extends the definitions and the implementation to detect redundant tests for advice and intertype methods. Second, Aspectra integrates the redundant-test detection, based on dynamic analysis, with a static analysis of AspectJ programs [25].

This paper makes the following main contributions:

- We propose Aspectra, a framework for detecting redundant unit tests for aspect-oriented programs; to the best of our knowledge, this is the first such framework.
- We exploit the results of static analysis to effectively improve the precision of Aspectra in dynamically detecting redundant tests.
- We present an implementation of Aspectra for detecting redundant unit tests for advice, advised methods, and intertype methods.
- We describe our experience in applying Aspectra to 12 AspectJ programs from a variety of sources. The experience shows that Aspectra 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, and the AspectJ compiler [1, 13]. Our implementation for detecting redundant tests operates on AspectJ programs, but the underlying ideas apply to the general class of aspect-oriented languages such as Hyper/J [29].

AspectJ is a seamless, aspect-oriented extension to Java. AspectJ adds to Java several new concepts and associated 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 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. An aspect can also use an intertype declaration to add a public or private method, field, or interface implementation declaration into a class.

The AspectJ compiler ensures that the base and aspect code run together in a properly coordinated fashion [1, 13]. 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. After aspect weaving, these base classes are then called woven classes and the methods in these classes are called advised methods.

In particular, the AspectJ compiler compiles 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. 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, 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 piece of advice is invoked on the aspect object.

Both pieces of before and after 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 compiles each intertype method declaration in the aspect 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 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.

3 Example

We next illustrate how Aspectra determines redundant tests for an AspectJ program. We use a simple integer stack example adapted from Rinard *et al.* [25]. 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 the implementation of the iterator class. Figure 3 shows four aspects for the stack class: `NonNegative`, `NonNegativeArg`, `Instrumentation`, and `PushCount`.

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 increases the static `count` field defined in the aspect. The advice is executed after any call to `Stack`’s `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 `Stack`’s constructor. The `PushCount` aspect is another version for counting the number of times a `Stack`’s `push` method is invoked on an object since its creation. The aspect contains one intertype declaration that declares a `count` field for the `Stack` class. The aspect contains another intertype declaration that declares an `increaseCount` method for the `Stack` class. The method increases the `count` intertype field of `Stack`. The aspect also contains a piece of around advice that invokes the `Stack`’s `IncreaseCount` in-

---

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

2 We declare this intertype method as public for illustration purpose. Then a client can invoke the `increaseCount` method to increase `count` without invoking `push`. 
Detecting Redundant Unit Tests for AspectJ Programs

```java
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); }
}

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;
    }
}
```

**Fig. 1.** An integer stack implementation

```java
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;
    }
}
```

**Fig. 2.** Stack Iterator

After we use the AspectJ compiler [1, 13] to compile and weave each of four aspects, we can use existing Java test-generation tools, such as Parasoft Jtest 4.5 [23], 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):**
```
Stack s1 = new Stack();
s1.push(3);
s1.push(2);
s1.pop();
s1.push(5);
```

**Test 2 (T2):**
```
Stack s2 = new Stack();
s2.push(3);
s2.push(5);
```

**Test 3 (T3):**
```
Stack s3 = new Stack();
s3.push(3);
s3.push(2);
s3.pop();
s3.pop();
```
To determine redundant tests for advised methods, Aspectra 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, push adds 5 to the previous stack. We call two method executions equivalent if they are invocations of the same method on equivalent states. Aspectra 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
Detecting Redundant Unit Tests for AspectJ Programs

For example, Aspectra 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, Aspectra 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*. Aspectra 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. Aspectra 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 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, Aspectra 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) at the beginning of the execution. We call this state *intertype-entry state*. Aspectra 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. Aspectra 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 equivalent to the intertype method execution generated by \( s1.push(5) \). The values of the count field are different at the entry of these two intertype method executions.

The advice execution in the PushCount aspect is characterized by the advice’s arguments: the target Stack object. Aspectra can detect that Test 3 is redundant but Test 2 is not redundant for the advice. However, Aspectra can additionally use R-

---

3 There is no receiver object for the intertype method because the intertype method in the aspect class is static.
nard et al.’s static analysis [25] to determine that the advice or the intertype method increaseCount reads only the intertype field count. In other words, only the value of the count field can affect the behavior of the advice or the intertype method. Then at the beginning of the advice execution or intertype method execution, the representation of the state is reduced to contain only the value of the count field instead of the whole Stack state. Based on this improved representation of advice-entry state or intertype-entry state, Aspectra can detect that both Test 2 and Test 3 are redundant for advice or intertype methods.

4 Redundant-Test Detection for AspectJ Programs

We consider three levels 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 next instantiate the definition for each of the three levels, describing how to determine the appropriate states for advised methods, advice, and intertype methods. We finally present how to exploit the results of a static analysis to improve the precision of redundant-test detection by focusing on relevant state information.

To detect redundant tests for advised methods, advice, and intertype methods, we have developed Aspectra, an extension of our previous framework [31], which can be viewed as detecting redundant tests only for advised methods. The input to Aspectra is a class under test and a set of methods under test; Aspectra 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, Aspectra 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 based on class bytecode, e.g., Parasoft Jtest 4.5 [23].
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, using the Aspectra framework, for the three levels 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 Aspectra 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 will behave the same. In particular, this means that Aspectra 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 Aspectra. Aspectra can collect 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 Aspectra 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.

Aspectra represents a method-entry state using the WholeState technique from our previous framework [31]. 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, Aspectra traverses and collects the values of (some) fields reachable from these objects. Next section presents which fields Aspectra collects for each of the three levels of units.

During the traversal, Aspectra performs a linearization [31] on the collected field values of reference type. Our linearization is similar to the standard Java serialization [28]: 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 [31]. What the linearization achieves is that states can be compared as sequences.

The state representation of a method-entry state is essentially a sequence of integers; comparing two states is reduced to comparing two sequences of strings. We denote with \( \text{linearize}(s) \) the state representation of a method-entry state \( s \).

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

\[
\text{linearize}(s_1) = \text{linearize}(s_2).
\]

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.** Two method executions \( \langle m, s \rangle \) and \( \langle m', s' \rangle \) are equivalent iff \( m = m' \) and \( \text{linearize}(s) = \text{linearize}(s') \).
Set minimization(Set origTests) {
    Set methodExecs = emptySet;
    Set nonRedundantTests = emptySet;
    foreach (Test t in origTests) {
        Set currentMethodExecs = runAndCollect(t);
        if !(currentMethodExecs subset methodExecs) {
            nonRedundantTests = nonRedundantTests add t;
            methodExecs = methodExecs union currentMethodExecs;
        }
    }
    return nonRedundantTests;
}

Fig. 4. Pseudo-code of the test minimization algorithm.

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 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 4 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 collects 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 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 [14]. No matter in what order the tests in \( S' \) are executed, the total number of detected nonequivalent method executions or object states remains the same. Since the inspection effort for automatically generated tests should focus on nonequivalent method executions, it is practical for our tool to use the test-execution order enforced by JUnit instead of searching for the optimal test-execution order.

4.2 Collecting States for Units

The above 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 types of units, advice and intertype methods. For both of them, we treat the aspect class as the class under test.

**Collecting State for Advice** We first discuss the specifics of methods that represent pieces of advice and then illustrate the special treatment of the JoinPoint arguments for advice.

The receiver object of advice is an aspect object (obtained by calling `aspectOf`). We treat inlined around advice in the base class as special advice. There is no receiver object for inlined around advice because it is a static method and the receiver object of its advised method is turned into the first argument of the static method. 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 method-entry states discussed in Section 4.1.

The body of a piece of advice can use three special variables—`thisJoinPoint`, `thisJoinPointStaticPart`, and `thisEnclosingJoinPointStaticPart`—to discover both static and dynamic information about the current join point [1, 13]. The AspectJ compiler detects which special variables are referred to within the body of the advice and extends the signature of the advice with corresponding arguments for these special variables. For example, the `NonNegativeArg` aspect shown in Figure 3 invokes

```java
thisJoinPoint.getArgs()
```
to retrieve the arguments of the current join point and invokes

```java
thisJoinPoint.getSignature().toShortString()
```
to get the method signature name associated with the current join point. The AspectJ compiler extends the signature of the advice with one additional argument:

```java
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, Aspectra needs to carefully collect the state to avoid collecting more information than 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

```java
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 JoinPoint 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 \texttt{thisJoinPoint.getArgs()} is an object ar-
ray 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 \texttt{thisJoinPoint.getSignature()} is an object with 
an AspectJ-library type \texttt{org.aspectj.lang.Signature}. We do not traverse and col-
lect the object fields of this signature object because it is an AspectJ-library-type object. 
Then \texttt{toString()} is invoked on this signature object and the method return is a 
\texttt{String}, containing the short-form name of the method signature. We also collect this 
string as part of the advice-entry state.

**Collecting State for Intertype Methods** In AspectJ, all intertype declarations are com-
plied into 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 method-entry states discussed in Section 4.1.

We use the test minimization algorithm, shown in Figure 4, for minimizing a test 
suite for testing intertype methods. In our implementation, we combine the redundant-
test detection for intertype methods and advice by treating intertype methods as a spe-
cial type of advice. In the rest of this paper, redundant-test detection for advice refers to 
redundant-test detection for both advice and intertype methods unless stated otherwise. 
However, in test generation for AspectJ programs, we shall still distinguish between in-
tertype methods and advice. When the AspectJ compiler weaves intertype methods into 
the base class, these intertype methods can become a part of the base-class interface. 
Therefore, the Java test generation tools based on bytecode cannot directly generate 
method inputs that exercise advice but can directly generate method calls that exercise 
intertype methods.

**4.3 Improving Detection with Static Analysis**

We next discuss how a static analysis can help Aspectra detect a larger number of re-
dundant tests. We have integrated Aspectra with Rinard \textit{et al.}'s static analysis [25] that 
detects read- and write-sets for AspectJ programs. This static analysis enables Aspectra 
to collect only the relevant parts of the object graphs for each method-entry state, which 
results in smaller method-entry states, a larger number of equivalent states and method 
executions, and finally a larger number of redundant tests.

Rinard \textit{et al.}'s analysis detects several facts about each unit (advised method, ad-
vice, or intertype method). The most relevant fact for Aspectra is the read-set for each 
unit: the set of fields that the unit may read during its executions (the unit may addi-
tionally write these fields). The analysis is conservative: all executions of the unit do 
not necessarily read all the fields from the read-set, but no execution reads a field that is 
not in the read-set. The analysis builds on a combined pointer and escape analysis [26] 
that can detect objects local to the unit: the unit allocates these objects, uses them lo-
cally, but never returns them as a result. Detecting local objects enables the analysis to 
produce more precise facts: the read-set for a unit does not include the effects on local
objects, because they are not visible outside of the unit. The granularity of the analysis is at the level of class fields. The read-set consists of per-class, not per-object, fields.

Aspectra determines the relevant state for a unit based on the read-sets computed by the static analysis. While collecting the method-entry states, Aspectra collects only the values of the fields that are in the read-set, essentially projecting the whole state on only the relevant fields. If the values of these fields are the same for two method-entry states, then these states are equivalent. The values of the other fields are irrelevant, since no execution could depend on them.

5 Experience

We have collected 12 AspectJ benchmarks from a variety of sources (Section 5.1). We have implemented the techniques of Aspectra (Section 4) and an existing related technique for comparison: the concern-based testing technique [27, 37] (Section 5.2). We applied these techniques to the collected benchmarks. The results show that the percentages of detected redundant tests are usually in increasing order for the concern-based testing technique, detection for advised methods, advice, and advice with static analysis (Section 5.3). The results suggest that our new techniques perform better than the existing concern-based testing technique, fewer tests need to be inspected for testing advice than testing advised methods, and static analysis can further reduce the inspection effort by detecting more redundant tests for advice. We also discuss some issues of Aspectra (Section 5.4).

5.1 Benchmarks

Our benchmarks include most of the programs used by Rinard et al. [25] in evaluating their classification system for aspect-oriented programs. The benchmarks also include most of the programs used by Dufour et al. [8] in measuring performance behavior of AspectJ programs. Our benchmarks do not include several AspectJ programs from these two sources primarily because these several programs are concurrent programs (our Aspectra framework works only on sequential programs) or GUI applications (GUI applications are not suitable subjects for Jtest’s automated test generation). Our benchmarks also include one of the aspect-oriented design pattern implementations by Hannemann and Kiczales [9]. Our benchmarks do not include other design pattern implementations because they primarily use the feature of intertype declarations and this feature has already been used by some of our benchmarks.

Table 1 lists the benchmarks that we use. The first four benchmarks (NonNegative, NonNegativeArg, Instrumentation, and PushCount) are the aspect examples in Figure 3. The NullCheck benchmark is an AspectJ program used by Asberry to detect whether method calls return null [3]. Following Rinard et al. [25], we refer to these

---

4 The AspectJ programs used by Dufour et al. [8] can be obtained from http://www.sable.mcgill.ca/benchmarks/.

5 Hannemann and Kiczales’s design pattern implementations can be obtained from http://www.cs.ubc.ca/~jan/AODPs/.
first five benchmarks as basic aspects. The Telecom benchmark is an example available with the AspectJ distribution [1]. It simulates a community of telephone users. The BusinessRuleImpl benchmark comprises two aspects of business rules for a banking system, which were used as examples in Section 12.5 of [17]. The StateDesignPattern benchmark had been implemented using AspectJ by Hannemann and Kiczales [9]. The DCM benchmark was implemented using AspectJ by Hassoun et al. [12] to validate their proposed dynamic coupling metric (DCM) [11]. The ProdLine benchmark was implemented using intertype declarations by Lopez-Herrejon and Batory for product lines of graph algorithms [21]. The Bean benchmark was used as an example by the AspectJ primer on aspectj.org. It enhances a class with the functionality of Java beans. The LoD benchmark was implemented by Lieberherr et al. to check the Law of Demeter [19]. It includes one checker for object form and the other one for class form. We focus on testing the checker for object form. Because the DCM and LoD benchmarks as well as the first five benchmarks do not come with base classes, we use the Stack class (shown in Figure 1) or its adapted version as their base classes.

5.2 Implementations

We have implemented the techniques of Aspectra in Section 4 for AspectJ programs by modifying Rostra, our previous tool for detecting redundant object-oriented unit tests [31]. We reuse Rostra to detect redundant tests for advised methods. During class loading time, we dynamically determine whether a class is an aspect class by inspecting the names of its methods, because the AspectJ compiler gives special names for advice. We also similarly detect inlined around advice in the base class based on their names. When detecting redundant tests for advice, we treat the identified aspect classes as the classes under test and the identified advice as the methods under test. To improve the redundant-test detection for advice, we first use Rinard et al.’s static analysis tool [25] to analyze AspectJ programs. Their tool produces a list of fields that could be potentially read or written by each advice method or advised method. Our tool exploits this static information to remove irrelevant state information when collecting advice-entry states.

There exists one concern-based testing technique [27, 37] that is most related to Aspectra. The concern-based testing technique selects a test if this test covers at least one piece of advice (no matter whether the input to the advice has been exercised before). We also implemented the concern-based testing technique in order to quantitatively compare it with our new techniques.

For each benchmark, we first feed its woven class bytecode to Jtest 4.5 [23] to generate tests. Jtest allows the developer 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.

We run these generated tests with the tool that we implemented for the concern-based testing technique. The tool reports the percentage of unselected tests (Column 4), which corresponds to the percentage of redundant tests in the context of our new tech-

\[\text{Although we could similarly apply static analysis to improve the redundant-test detection for advised methods, we focus on improving the detection for advice in our evaluation.}\]
Detecting Redundant Unit Tests for AspectJ Programs

Table 1. Results of applying redundant-test detection on Jtest-generated tests

<table>
<thead>
<tr>
<th>AspectJ program</th>
<th>tests</th>
<th>bcov</th>
<th>%r-tests</th>
<th>ne-methexec</th>
<th>ne-objstates</th>
</tr>
</thead>
<tbody>
<tr>
<td>NonNegative</td>
<td>44</td>
<td>6/7</td>
<td>6.8</td>
<td>72.7</td>
<td>90.9</td>
</tr>
<tr>
<td>NonNegativeArg</td>
<td>44</td>
<td>8/9</td>
<td>6.8</td>
<td>72.7</td>
<td>90.9</td>
</tr>
<tr>
<td>Instrumentation</td>
<td>44</td>
<td>4/4</td>
<td>0.0</td>
<td>72.7</td>
<td>84.1</td>
</tr>
<tr>
<td>PushCount</td>
<td>94</td>
<td>8/8</td>
<td>0.0</td>
<td>70.2</td>
<td>77.7</td>
</tr>
<tr>
<td>NullCheck</td>
<td>45</td>
<td>4/6</td>
<td>60.0</td>
<td>71.1</td>
<td>91.1</td>
</tr>
<tr>
<td>Telecom</td>
<td>798</td>
<td>19/28</td>
<td>0.0</td>
<td>95.2</td>
<td>98.5</td>
</tr>
<tr>
<td>BusinessRuleImpl</td>
<td>439</td>
<td>14/21</td>
<td>50.1</td>
<td>94.1</td>
<td>97.7</td>
</tr>
<tr>
<td>StateDesignPattern</td>
<td>129</td>
<td>13/17</td>
<td>0.0</td>
<td>48.8</td>
<td>36.4</td>
</tr>
<tr>
<td>DCM</td>
<td>44</td>
<td>57/91</td>
<td>0.0</td>
<td>72.7</td>
<td>0.0</td>
</tr>
<tr>
<td>ProdLine</td>
<td>474</td>
<td>80/294</td>
<td>0.0</td>
<td>86.5</td>
<td>90.5</td>
</tr>
<tr>
<td>Bean</td>
<td>1895</td>
<td>18/19</td>
<td>0.0</td>
<td>69.9</td>
<td>73.7</td>
</tr>
<tr>
<td>LoD</td>
<td>44</td>
<td>8/133</td>
<td>18.2</td>
<td>68.2</td>
<td>18.2</td>
</tr>
</tbody>
</table>

Table 1. Results of applying redundant-test detection on Jtest-generated tests

niques, and the number of advice executions (Column 8). The results of the concern-based testing technique are put in the columns with the title of “con”.

We then run these generated tests with our tool to detect redundant tests using our new techniques. For each of advised methods, advice, and advice with static analysis, the tool reports the percentage of redundant tests (Columns 5, 6 and 7), the number of nonequivalent method or advice executions (Columns 9, 10, and 11), and the number of nonequivalent class or aspect object states (Columns 12 and 13, for advised methods and advice only). The results for advised methods, advice, and advice with static analysis are put in the columns with the titles of “meth”, “adv”, and “sta”, respectively. The LoD benchmark does not have the results for advice with static analysis, because Rinard et al.’s static analysis tool [25] cannot successfully analyze the LoD benchmark caused by the version incompatibility of Java library.

The third column of Table 1 shows the coverage of the branches within aspect classes achieved by tests generated by Jtest (we have adapted Hansel [10] to measure branch coverage of aspect classes in the bytecode level). To assure the correctness of our implementation, we measured the aspect branch coverage achieved by test suites minimized with different techniques and found that the coverage was the same as the one achieved by their original test suites.

5.3 Results

For each benchmark, we first discuss the characteristics of aspect implementations and the results of our redundant-test detection. We then summarize the comparison of different techniques.

Basic Aspects. Our detection technique for advised methods detects the same percentage of redundant tests on the first three benchmarks in 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 ini-
tializer method <clinit> into the Stack class for preparing the joinpoint reflection
within NonNegativeArg. In general, when a base class is woven with different as-
pects, running the same test suite on the woven class might produce different numbers
of redundant tests, nonequivalent method executions, or nonequivalent object states for
advised methods.

The PushCount aspect has lower percentage of redundant tests, more nonequiva-
lent method executions, and more nonequivalent object states than the first three aspects,
because PushCount declares an intertype field and an intertype method for Stack.

The NullCheck benchmark uses around advice for those methods whose returns
are not void. To provide a base class for NullCheck, we adapt the Stack class in
Figure 1 by changing the int type to Integer. Then both the pop and iterator
methods of Stack are advised by NullCheck. 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 all the basic aspects (the first five aspects in Table 1), our
techniques detect more redundant tests for advice than for advised methods.

**Telecom.** One key base class in the Telecom benchmark is 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 technique for advice detects only two nonequivalent aspect states, which
are empty states. 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 objects of these method calls. Therefore,
our technique for advice 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 for the call of Connection’s constructor and the other one for the call
of Connection’s drop(). 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 (the receiver objects or
arguments) are visible and usable to advice executions. Our techniques detect more
redundant tests for advice than for advised methods.

**BusinessRuleImpl.** An MinimumBalanceRuleAspect defines a piece of before advice
for the execution of Account’s debit method. Another OverdraftProtection
RuleAspect defines another piece of before advice for the execution of Account’s
debit method7. Neither aspect declares any object field for itself; our technique for
advice detects only two nonequivalent aspect states exercised by the generated tests.

---

7 The original version in the book [17] requires the debit execution for the advice to be in-
voked underneath a method in a CheckClearanceSystem class; we comment this con-
straint out for the generated tests to exercise both aspects.
The arguments to both pieces of advice are the target Account object and the argument (the withdrawal amount) of the debit. Our techniques detect more redundant tests for advice than for advised methods.

**StateDesignPattern.** 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 our techniques detect fewer redundant tests for advice than for advised methods.

**DCM.** The DCM aspect uses around and after advice for the execution of a constructor or method in the base class. The aspect defines several static fields to keep track of call depth, the class name of the current caller, etc. It also uses a hash table to compute the dynamic coupling metric. 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 the static fields defined in the DCM aspect store the information 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.

**ProdLine.** The base classes of the ProdLine benchmark 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. Because intertype fields are declared in the base class and no fields are defined in the aspect classes, our technique for advice detects six nonequivalent aspect object states. These six aspect object states are the empty states of six different aspects.

**Bean.** The base class of the Bean benchmark is the Point class with two instance fields for coordinates: x and y. The BoundPoint aspect declares an intertype field for Point. This new field has the type of PropertyChangeSupport. The aspect also declares five intertype methods related to the new field. Then the aspect uses around advice for the execution of the setX or setY method of Point. Our technique for advice detects slightly more redundant tests than for advised methods. Because there is no extra field defined on the BoundPoint aspect and no non-static method of BoundPoint is invoked, our technique for advice detects that no aspect object states are exercised.
**LoD.** One main component of the LoD object form checker is the Check aspect. The Check aspect declares two hash-map-type fields for collecting the places that violate the Law of Demeter. The aspect declares two pieces of after advice for checking the method calls. There are two other aspects Percflow and Pertarget for collecting calling context through the use of perflow, pertarget, and cflow. Because our monitoring of the Percflow and Pertarget aspect executions can throw StackOverflowException, we configure our techniques to detect redundant tests for only the Check aspect. Although our technique for advice detects only two nonequivalent Check aspect object states, our technique for advised methods detects 55 nonequivalent method executions. By inspecting the trace, we found that 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 detects fewer redundant tests for advice than for advised methods.

**Summary.** The existing concern-based testing technique detects no redundant tests (selects all tests) for eight benchmarks. For the remaining four benchmarks, some tests never cover any advice and get selected as redundant tests by the existing technique. In general, our new techniques can detect more redundant tests than the existing technique.

Jtest generates a relatively small number of method arguments for the methods of the class under test and generates many different combinations of method call sequences. Our technique for advised methods can often detect a high percentage of redundant tests among the tests generated by Jtest. This phenomenon has been observed in the experiments of evaluating Rostra [31] and the experiments showed that removing these redundant tests does not decrease the fault detection capability and structural coverage achieved by the test suite. Furthermore, our technique for advice can detect an even higher percentage of redundant tests. 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 or advice executions and class or aspect object states; the developer 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. Three interesting exceptional cases are the StateDesignPattern, DCM, and LoD benchmarks; for these three benchmarks, our techniques detect fewer redundant tests for advice than for advised method.

We can use static analysis to further detect more redundant tests for advice of seven benchmarks. Among the remaining four benchmarks (excluding the unanalyzable LoD benchmark), three are simple basic aspects. The advice in the NonNegative benchmark reads the whole state of Stack; therefore, static analysis cannot help remove any part of the state. The advice in the NonNegativeArg or Instrumentation benchmark does not read any part of the state of Stack and the arguments of the advice do not include the receiver object of its advised method; therefore, static analysis does not help detect more redundant tests either. The aspect in the DCM benchmark keeps track of extensive runtime calling information and even static analysis cannot help detect any redundant test. In summary, our experience shows that it is promising to exploit static analysis to improve redundant-test detection, especially for complex aspects.
5.4 Discussion

We expect that Aspectra can be applied to a wide range of AspectJ programs. Aspectra can substantially reduce the size of generated tests for inspection when specifications are absent, a common case in practice. We expect that, for most AspectJ programs, testing for advice would require less inspection effort than testing for advised methods. Indeed, based on our experience, we found that there were cases that more inspection effort is needed for testing advice than for testing advised methods. Aspectra still provides value for these types of AspectJ programs; Aspectra can improve developer’s understanding of the behavior of aspects and draw their attention on the aspects’ dominating behavior.

Aspectra are primarily based on dynamic analysis; in order to detect redundant tests for advice or advised methods, Aspectra needs to run the generated tests. Based on our experience with Jtest and these 12 benchmarks, Jtest’s test generation is relatively expensive for large programs but the execution of the generated tests is usually cheap and the runtime overhead incurred by our dynamic analysis is reasonable. Although the goal of the redundant-test detection in this work is to reduce the inspection effort, Aspectra could potentially open up opportunities of generating tests for aspect-oriented programs. Our recent work [30, 32] extends Rostra to efficiently generate tests for Java programs. In Section 7, we lay out directions that we plan to pursue in generating tests for AspectJ programs.

6 Related Work

Souter et al. [27] developed a concern-based testing technique. The technique identified the code associated with a particular maintenance task (referred as a concern) and performed testing tasks with respect to the concern. They 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 technique 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. [37] used the same technique for selecting relevant tests for an aspect. When testing an aspect using Aspectra, we select a test if the test covers the aspect and the input to the aspect is different from any previously exercised input. Therefore, Aspectra selects fewer tests with respect to an aspect than these two previous techniques (shown by our experience in Section 5) but preserves the same quality of tests selected by these techniques for testing the aspect.

Xu et al. [33] presented a state-based testing approach for aspect-oriented software. The basic idea of the approach is to extend the existing FREE (Flattened Regular Expression) state model, which originally proposed for testing object-oriented programs, to model issues related to aspects in an aspect-oriented program. Based on the model, they developed two techniques for testing aspect-oriented programs based on transforming an aspect state model to a transition tree to generate some test cases and the aspect flow graph of an aspect-oriented program, respectively. Their work focuses on testing
aspect-oriented programs based on some abstract state models, whereas Aspectra 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 [23].

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 Aspectra, it does not provide a concrete method for testing aspect-oriented programs.

Zhao [34, 35] 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 Aspectra focuses on detecting redundant unit tests for AspectJ program based on object states, in order to reduce the unit testing cost.

Krishnamurthi et al. [16] 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. Aspectra also implicitly models a piece of advice as a state machine and focuses on testing aspects.

Zhao and Rinard [36] 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) [18], 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 effort. 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, advice executions, or intertype method executions proposed in our approach.

Rajan and Sullivan [24] presented an approach to expressing and automating test adequacy criteria relative to crosscutting concerns using aspect-oriented languages. Their approach represents tester intentions within source code in an explicit and abstract way. They also provided a white-box join point model and a generalized action framework to support white-box testing tools. Their work focuses on using aspect-oriented languages to support general and automated test adequacy analysis, whereas Aspectra focuses on how to detect redundant tests for AspectJ programs.

Rinard et al. [25] proposed a classification system for aspect-oriented programs. The system characterizes the interactions between advice and advised methods in terms of field accesses. They also developed a static analysis tool for automatically classifying interactions between advice and advised methods. Their classification system and analysis can help developers structure their understanding of the aspect-oriented programs. By using dynamic analysis, Aspectra complements their system and static analysis. Besides reporting redundant tests, Aspectra also reports the number of different inputs to advice or advised methods exercised by the same tests. Comparing these numbers for
Detecting Redundant Unit Tests for AspectJ Programs

advice and advised methods offers new insights into the behavior of aspect-oriented programs. For example, by inspecting the last two columns of Table 1, developers can easily identify the special interaction behavior of the state design pattern and DCM benchmarks among the 12 benchmarks in Section 5. Rinard et al.’s static analysis can report whether one field in aspects is written after reading some fields in the base class, but cannot capture the quantitative information about how substantially the field values in the aspect classes are changed when the field values in the base classes are changed. Aspectra provides this useful dynamic information to complement the results of their static analysis.

7 Conclusion and Future Work

We have proposed Aspectra, a novel framework for detecting redundant unit tests for AspectJ programs. Redundant tests are defined for three types of units: pieces of advice, advised methods, and intertype methods. We have formally defined inputs to either type of units based on object states. Aspectra has extended our previous Rostra framework [31], which detects redundant tests for advised methods, to detect redundant tests for advice and intertype methods. In addition, Aspectra has also exploited the results of static analysis to determine the relevant inputs to the units, effectively improving the precision of the redundant-test detection. 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 or intertype methods, we can use the test generation techniques for Java programs by avoiding redundant tests [30]. In test generation for an advised method or intertype methods, we have direct controls on choosing which inputs to exercise the method. We also have direct controls to choose which inputs to exercise the method in testing generation for an intertype 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 can use Rinard et al.’s static analysis [25] 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.

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 his help in our use of his group’s static analysis tool and his valuable feedback on an earlier version of this paper.

References