A Framework and Tool Supports for Generating Test Inputs of AspectJ Programs

Tao Xie
Department of Computer Science
North Carolina State University
Raleigh, NC 27695
xie@csc.ncsu.edu

Jianjun Zhao
Department of Computer Science & Engineering
Shanghai Jiao Tong University
Shanghai 200240, China
zhao-jj@cs.sjtu.edu.cn

ABSTRACT

Aspect-oriented software development is gaining popularity with the wider adoption of languages such as AspectJ. To reduce the manual effort of testing aspects in AspectJ programs, we have developed a framework, called Aspectra, that automates generation of test inputs for testing aspectual behavior, i.e., the behavior implemented in pieces of advice or intertype methods defined in aspects. To test aspects, developers construct base classes into which the aspects are woven to form woven classes. Our approach leverages existing test-generation tools to generate test inputs for the woven classes; these test inputs indirectly exercise the aspects. To enable aspects to be exercised during test generation, Aspectra automatically synthesizes appropriate wrapper classes for woven classes. To assess the quality of the generated tests, Aspectra defines and measures aspectual branch coverage (branch coverage within aspects). To provide guidance for developers to improve test coverage, Aspectra also defines interaction coverage. We have developed tools for automating Aspectra's wrapper synthesis and coverage measurement, and applied them on testing 12 subjects taken from a variety of sources. Our experience has shown that Aspectra effectively provides tool supports in enabling existing test-generation tools to generate test inputs for improving aspectual branch coverage.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools (e.g., data generators, coverage testing)*

General Terms

Experimentation, Measurement, Reliability, Verification

Keywords

Aspect-oriented software development, aspect-oriented programs, AspectJ, software testing, test generation, coverage criteria, coverage measurement

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
AOSD 06, March 20–24, 2006, Bonn, Germany
Copyright 2006 ACM 1-59593-300-X/06/03 ...$5.00.

1. INTRODUCTION

Aspect-oriented software development (AOSD) is a new technique that improves separation of concerns in software development [9, 18, 22, 30]. AOSD makes it possible to modularize crosscutting concerns of a software system, thus making it easier to maintain and evolve. Research in AOSD has focused mostly on the activities of software system design, problem analysis, and language implementation. Although it is well known that testing is a labor-intensive process that can account for half the total cost of software development [8], research on testing of AOSD, especially automated testing, has received little attention.

Although several approaches have been proposed recently for testing aspect-oriented programs [4, 35, 37, 38], none of these approaches is able to provide a framework for automated generation of test inputs for AspectJ programs. AOSD can lead to better-quality software, but it 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 during the system development. In addition, Aspect-oriented programming can introduce specific (and hard to detect) errors that ordinary object-oriented programming is not subject to. As a result, software testing remains an inevitable and important task in AOSD.

Aspect-oriented programming languages, such as AspectJ [18], introduce some new language constructs (such as join points, advice, intertype declarations, and aspects) to the common object-oriented programming languages, such as Java. The behavior of an aspect in AspectJ programs can be categorized into two types [23]: *aspectual behavior* (behavior implemented in pieces of advice) and *aspectual composition behavior* (behavior implemented in pointcuts for composition between base and aspectual behavior).

When we treat an aspect as a unit and intend to test its aspectual behavior, *unit tests* for an aspect are created to test in isolation pieces of advice defined in the aspect. However, it is often difficult to manually or automatically construct the aspect's execution context in unit tests. When we intend to test aspectual composition behavior related to an aspect, *integration tests* for the aspect are created to test interaction or composition between the aspect class and the affected classes. These integration tests can consist of invocations of those methods affected by the aspect. These invocations eventually exercise the interaction between the aspect class and the affected classes by invoking pieces of advice from the advice-call sites inserted within the affected classes.

Our research focuses on automatic generation of test inputs that test aspectual behavior, an important type of an aspect's behavior. We leave the issue on automatic generation of test inputs for aspectual composition behavior for our future work. Specifically, we

propose Aspectra, a novel framework for generating test inputs to exercise aspectual behavior. Given aspects to be tested, developers can construct base classes that the aspects can be woven into to produce woven classes. We can view these base classes as providing scaffoldings necessary to drive the aspects. Aspectra develops a wrapper-synthesis technique to address aspect weaving issues in test generation (by providing visibility of woven methods to test-generation tools and avoiding unwanted weaving). Given a woven class, Aspectra automatically synthesizes a wrapper class for the woven class and then feeds the wrapper class to our test-generation tool based on state exploration [33, 34]. In order to assess the quality of generated tests, we define and measure aspectual branch coverage, which characterizes branch coverage within aspect code. Sometimes initially generated test inputs for base classes (constructed by developers) may not be sufficient to achieve good aspectual branch coverage. To guide developers to improve test coverage, we define interaction coverage that measures the interactions among four types of methods in AspectJ programs (which will be explained in Section 4.1): advised methods (defined in base classes), advice, intertype methods, and public non-advice methods (defined in aspects). These measurement results guide developers to improve the base-class construction and test generation.

This paper makes the following main contributions with the proposal of the Aspectra framework.

- We develop a wrapper-synthesis technique that prepares woven classes to be given to test-generation tools; The synthesized wrapper classes provide a clean interface between the program under test and test-generation tools.We implement a tool to automate the wrapper synthesis.
- We leverage existing tools for testing Java programs to generate tests for AspectJ programs.
- We define and measure branch coverage within aspect code. We also classify four types of methods in AspectJ programs and measure the interactions among them. We implement tools to automate these measurements. We provide guidelines for developers to use these measurement results to improve test coverage.
- We describe our experience in applying Aspectra to 12 AspectJ programs from a variety of sources. The experience shows that Aspectra provides effective tool supports to generate test inputs for increasing structural coverage of aspect code.

The rest of the paper is organized as follows. Section 2 briefly introduces AspectJ. Section 3 presents an example that we shall use to illustrate our approach. Section 4 presents our Aspectra framework. Section 5 describes our implementation for Aspectra. Section 6 presents our experience in applying Aspectra on various AspectJ programs. Section 7 discusses related work and Section 8 concludes.

2. ASPECTJ

Aspectra generates test inputs for AspectJ programs. We next introduce background information on AspectJ [1]. Although we present Aspectra in the context of AspectJ's ajc compiler [1,16], the underlying ideas are applicable to other AspectJ compilers [2, 7].

AspectJ adds to Java some new concepts and associated constructs including join points, pointcuts, advice, intertype declarations, and aspects. The *join point* in AspectJ is an essential concept in the composition of an aspect with other classes. It 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. A *pointcut* is a set of joint points that optionally expose some of the values in the execution of these joint points. AspectJ defines several primitive *pointcut designators* that can identify all types of join points. Pointcuts in AspectJ can be composed and new pointcut designators can be defined according to these combinations.

Advice is a method-like mechanism used to define certain code that executes *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.

Aspects are modular units of crosscutting implementation. Aspects are defined by aspect declarations, which have similar forms of class declarations. Aspect declarations may include pointcut, advice, and intertype declarations, as well as method declarations that are permitted in class declarations.

AspectJ compilers such as ajc [1, 16] use *aspect weaving* to compose the code of the base classes and the aspects to ensure that applicable advice runs at the appropriate join points. After aspect weaving, these base classes are then called *woven classes*.

During the weaving process, the ajc compiler [1, 16] compiles each aspect into an aspect class and each piece of advice in the aspect into a public method (called *advice method* in short as *advice*) 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. Then at appropriate locations of base classes, ajc inserts calls to compiled advice; methods (in base classes) that contain these call sites are *advised methods*. At each site of these inserted calls, a singleton object of an aspect class is first obtained by calling the static method `aspectOf` that is defined in the aspect class. Then a piece of advice is invoked on the aspect object. Note that the compilation of a piece of `around` advice [16] is more complicated than `before` or `after` advice. 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 complied `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 methods of the base class; no `around` advice method is created in the aspect class for this case.

The ajc compiler compiles each intertype field declaration in an aspect into a field in the base class and compiles each intertype method declaration in an 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. A wrapper method is inserted in the base class that invokes the actual method implementation in the aspect class. Moreover, all accesses to the fields inserted in the base class are through two public static wrapper methods in the aspect class for getting and setting field respectively. An aspect can also declare a public method that is not any type of advice. The ajc compiler also compiles it into a standard public Java method (called *public non-advice method*) in the aspect class. For more information about AspectJ weaving, refer to [16].

3. EXAMPLE

We use a simple integer stack example (adapted from Rinard *et al.* [28]) to illustrate our Aspectra framework throughout this paper. The example shows some common language features of AspectJ. Figure 1 shows the implementation of the stack class. This

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

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

Figure 1: Integer stack implementation

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

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

Figure 2: Stack iterator

class provides standard stack operations as public non-constructor methods: `push` and `pop`. The class also has one package-private method: `iterator` returns an iterator that can be used to traverse the items in the stack. The implementation of the iterator class is shown in Figure 2.

The stack implementation accommodates integers as stack items. Figure 3 shows three aspects that enhance the stack implementation. The `NonNegativeArg` aspect checks whether method arguments are nonnegative integers. The aspect contains a piece of advice that goes through all arguments of an about to be executed method to check whether they are nonnegative integers. The advice is executed before an execution of any method. The `NonNegative` aspect checks the property of nonnegative items: the aspect contains a piece of advice that iterates through all items to check whether they are nonnegative integers. The advice is executed before a call of a `Stack` method.

The `PushCount` aspect counts the number of times a `Stack`'s `push` method is invoked on an object since its creation. The aspect

```
aspect NonNegativeArg {
   before() : execution(* *.*(..)) {
      Object args[] = thisJoinPoint.getArgs();
      for(int i=0; i<args.length; i++) {
         if ((args[i] instanceof Integer) &&
          (((Integer)args[i]).intValue() < 0))
          throw new RuntimeException("negative arg of " +
            thisJoinPoint.getSignature().toShortString());
      }
    }
}

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

Figure 3: `NonNegativeArg`, `NonNegative`, and `PushCount` aspects

declares an intertype field `count` for the `Stack` class. The field keeps the number of times a `Stack`'s `push` method is invoked. The aspect declares a public intertype method `increaseCount` for the `Stack` class. The method increases the `count` intertype field of `Stack`. Note that we declare this intertype method as public for illustration purpose. Then a client can invoke the `increaseCount` method to increase `count` without invoking `push`. The aspect also contains a piece of `around` advice that invokes the `Stack`'s intertype method `increaseCount` declared in the aspect. The advice is executed around any execution of `Stack`'s `push` method. The `PushCount` aspect also defines a field `allStackCount` to record the number of times a `Stack`'s `push` method is invoked on any object. It defines a public method `getAllStackCount` for querying the value of `allStackCount`.

4. FRAMEWORK

We propose the Aspectra framework for generating test inputs to test aspectual behavior. Aspectra classifies the executions of four types of methods in an AspectJ program: advised methods defined in base classes, advice, intertype methods, and public non-advice methods defined in aspect classes (Section 4.1). Developers can construct base classes and weave aspects into base classes to produce woven classes, which can be fed to existing test-generation tools. To enable methods defined in aspects to be exercised during test generation, Aspectra develops a wrapper mechanism to prepare woven classes to be tested by existing test-generation tools (Section 4.2). Then Aspectra leverages our test-generation tool for Java

programs [33] to generate test inputs for AspectJ programs (Section 4.3). Because the initially generated tests may not be sufficient to cover aspectual behavior in aspects, Aspectra defines and measures aspectual branch coverage and interaction coverage to guide developers to improve the base-class construction and test generation (Section 4.4).

4.1 Method Executions of AspectJ Programs

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 in the sequence can eventually invoke some other methods, producing more method calls. 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. In a state representation for an object or multiple objects, we use the values of all the fields that are transitively reachable from the object(s) [33].

We classify the executions of four types of methods during the execution of an AspectJ program: the executions of *advised methods* defined in base classes, *advice*, *intertype methods*, and *public non-advice methods* defined in aspect classes. An example of a public non-advice method is `getAllStackCount` defined in the `PushCount` aspect (Figure 3). To test aspectual behavior, Aspectra focuses on testing the behavior exhibited by the last three types of methods defined in aspect classes. Note that when advised methods are executed, the last three types of methods defined in aspect classes may not be necessarily executed or covered. Our Aspectra framework helps increase the structural coverage of these methods defined in aspect classes.

4.2 Wrapper Synthesis

Given aspects, developers can construct appropriate base classes for the aspects and then use ajc to weave aspects into the constructed base classes to produce woven classes in the form of bytecode. Several automatic test-generation tools generate test inputs based on Java bytecode instead of source code. For example, both Parasoft Jtest [26] and JCrasher [10] generate random method sequences for the class under test based on its bytecode. Based on Java bytecode, our previous work developed Rostra [32, 33] and Symstra [34] for generating only method sequences that produce different inputs for methods under test. To generate test inputs for AspectJ programs, developers may simply feed woven classes (in the bytecode form) to these existing test-generation tools and use these tools to generate test inputs for the woven classes.

However, we need to address at least three major issues when we leverage these existing test-generation tools to generate test inputs for AspectJ programs:

- When a piece of advice is related to `call` join points, such as the advice in the `NonNegative` aspect, the existing test-generation tools cannot execute the advice during its test-generation process, because the advice is to be woven in call sites, which are not available before test generation.
- Although we can use ajc [1, 16] to weave the generated tests with the aspect classes in order to execute advice related to `call` join points, the compilation can fail when the interfaces of woven classes contain intertype methods and the generated test code invoke these intertype methods, such as the intertype method in the `PushCount` aspect. The test code cannot be compiled by ajc because ajc does not expect that the base class source files refer to intertype methods before the weaving process. In addition, weaving the generated test classes with the aspect classes could introduce unwanted advice into the test classes. For example, weaving test classes with the advice in the `NonNegativeArg` aspect introduces unwanted argument checking for methods defined in the test classes.
- Public non-advice methods in aspect classes cannot be exercised by the test inputs generated for woven classes, because woven classes do not have any call sites of these public non-advice methods. For example, no generated test inputs for the woven `Stack` class can exercise the public non-advice `getAllStackCount` method defined in `PushCount`.

Although here we do not intend to make a complete list of issues that can be encountered when testing full language features of AspectJ, the preceding issues are major ones that are encountered when we applied existing test-generation tools to a number of typical AspectJ programs. To address these major issues, Aspectra automatically synthesizes a wrapper class for each constructed base class for aspects and then the wrapper class is fed to existing test-generation tools. In particular, there are six steps for generating test inputs based on the wrapper mechanism (to simplify explanation, we focus on only one base class below):

1. Compile and weave the base class and aspects into class bytecode using ajc.
2. Synthesize a wrapper class for the base class based on the woven class and aspect bytecode.
3. Compile and weave the base class, wrapper class, and aspects into class bytecode using ajc.
4. Clean up unwanted woven code in the woven wrapper class.
5. Generate test inputs for the woven wrapper class using existing test-generation tools based on class bytecode.
6. Compile the generated test class into class bytecode using a Java compiler [5].

In the second step, we synthesize a wrapper class for the base class under test. Figure 4 shows the wrapper class synthesized for the `Stack` class (Figure 1) woven with the three aspects (Figure 3). In this wrapper class, we synthesize a wrapper method for each public method in the base class. This wrapper method invokes the public method in the base class.

In this wrapper class, we also synthesize a wrapper method for each public intertype method woven into the base class. This wrapper method uses Java reflection [5] to invoke the intertype method; otherwise, the compilation in the third step can fail because intertype methods are not recognized by ajc before compilation. For example, in the `PushCount` aspect, there is an intertype method `increaseCount` and we synthesize a wrapper method for it. In a similar way, we also synthesize a wrapper method for a public non-advice method in aspect classes. For example, in the `PushCount` aspect, there is a public non-advice method `getAllStackCount` and we create a public wrapper method for it in the wrapper class. These wrapper methods ensure that intertype methods or public non-advice methods of aspect classes are tested by existing test-generation tools.

In the third step, we use ajc to weave the wrapper class with the base class and aspects. This step ensures that the advice related to `call` join points is executed during test-generation process, because the invocations to the advice are woven into the call sites (within the wrapper class) of public methods in the base class.

In the fourth step, we need to clean up unwanted woven code in the woven wrapper class. For example, the wrapper method for the `push` method of `Stack` is also advised by the `execution` advice defined in the `NonNegativeArg` aspect. The advice is unwanted

```
public class StackWrapper {
   Stack s;
   public StackWrapper() {
      s = new Stack();
   }
   public boolean push(int i) {
      return s.push(i);
   }
   public int pop() {
      return s.pop();
   }

   public void increaseCount() throws Exception {
      /*  s.increaseCount(); */
      Class cls = Class.forName("Stack");
      Method meth = cls.getMethod("increaseCount", null);
      meth.invoke(s, null);
   }

   public int getAllStackCountPushCount() throws Exception {
      /*  PushCount ret1 = PushCount.aspectOf(); */
      /*  return ret1.getAllStackCount();        */
      Class cls = Class.forName("PushCount");
      Method meth1 = cls.getMethod("aspectOf", null);
      Object ret1 = (Object)meth1.invoke(null, null);
      Method meth2 = cls.getMethod("getAllStackCount", null);
      Integer ret2 = (Integer)meth2.invoke(ret1, null);
      return ret2.intValue();
   }
}
```

Figure 4: The wrapper class for `Stack`

by the wrapper method; otherwise, `push`'s arguments are checked twice during test generation, one time in the advised method and the other time in the wrapper method[1]. We scan the bytecode of the woven wrapper class and remove the woven code that are for advice related to `execution` join points. Note that we need to keep the woven code that is for advice related to `call` join points, because the woven code there is needed for covering the advice related to `call` join points.

In the fifth step, we feed the woven wrapper class to existing test-generation tools based on bytecode, such as Parasoft Jtest [26], JCrasher [10], Rostra [32,33], and Symstra [34]. These tools export generated tests to test code, usually as a JUnit test class [17]. The next section discusses how we use a combination of Jtest and Rostra to generate test inputs for wrapper classes.

In the final step, we use a Java compiler [5] to compile the exported test class. We do not use ajc to weave the exported test class with the wrapper class, base class, or aspects, because the weaving process can introduce unwanted woven code into the test class. For example, if we use ajc to weave the exported test class with `Stack` and the `NonNegativeArg` aspect, some methods defined in the test class are also advised by the `execution` advice defined in the `NonNegativeArg` aspect. The advice is unwanted by the methods defined in the test class.

4.3 Test-Input Generation

Among the four types of methods, the wrapper mechanism presented in the preceding section enables advised methods, intertype methods, and public non-advice methods to be directly exercised by generated test inputs. However, advice is not directly exercised but indirectly exercised through its advised method(s). We next illustrate how we leverage existing test-generation tools to generate test inputs to exercise wrapper methods (for advised methods, intertype methods, and public non-advice methods) in the interface of a wrapper class.

[1]We can avoid unwanted woven code in the woven `StackWrapper` if we rewrite the pointcut for `NonNegativeArg` to narrow down the scope to the methods of `Stack` rather than any method.

```
Set testgen(Set nonEqInitArgs, Set nonEqMethodArgs,
            int maxIterNum) {
  //generate object states after constructor calls
  Set newTests = new Set();
  foreach (args in nonEqInitArgs) {
    Test newTest = makeTest(args);
    newTests.add(newTest);
  }
  RuntimeInfo runtimeInfo = runAndCollect(newTests);
  Set frontiers = runtimeInfo.getNewNonEqObjStates();
  //combinatorial testing of object states and argument lists
  for(int i=1;i<=maxIterNum && frontiers.size()>0;i++) {
      Set newTestsForCurIter = new Set();
      foreach (objState in frontiers) {
        foreach (args in nonEqMethodArgs) {
          Test newTest = makeTest(objState, args);
          newTestsForCurIter.add(newTest);
          newTests.add(newTest);
        }
      }
      runtimeInfo = runAndCollect(newTestsForCurIter);
      frontiers = runtimeInfo.getNewNonEqObjStates().
  }
  return newTests;
}
```

Figure 5: Pseudo-code of Rostra's test-generation algorithm.

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

We use Parasoft Jtest 4.5 [26] to generate arguments for public methods of wrapper classes. Jtest uses symbolic execution [19] to generate method arguments to achieve structural coverage. By default, it generates method arguments for public methods of the class under test. Jtest exports its generated test inputs in the form of JUnit [17] test classes. Then we feed the Jtest-generated test classes to our Rostra tool [32,33], which uses method arguments to explore receiver-object state space. We next illustrate the algorithm of Rostra's test generation.

Rostra represents the state of an object with the values of all the fields that are transitively reachable from the object. A linearization algorithm [33] is used to linearize these values into a representation string. Two states are *equivalent* iff their state-representation strings are the same, and are *nonequivalent* otherwise. Rostra first executes Jtest-generated test classes and collects the exercised method arguments to form *method argument lists*, each of which is characterized by the method name, method signature, and the argument values for the method. Two argument lists are *equivalent* iff their method names, signatures are the same and the argument values are equivalent, and are *nonequivalent* otherwise.

Rostra's test generation is a type of combinatorial testing. It generates test inputs to exercise each possible combination of non-equivalent receiver-object states and non-equivalent method argument lists. The pseudo-code of the test-generation algorithm is presented in Figure 5.

Before running the algorithm, we first collect a set of non-equivalent constructor argument lists and method argument lists from the execution of Jtest-generated test classes. Then we feed the collected information as well as a (user-defined) maximum iteration number to the algorithm. In the algorithm, we first make a set of tests, each of which consists of a constructor call produced by using one of the non-equivalent constructor argument lists. Then we run these

tests and collect the non-equivalent receiver-object states produced by these constructor calls. We put these object states into a frontier set. Then we iterate each object state in the frontier set and each method argument list in the set of non-equivalent method argument lists. We make a test for each combination: the test invokes a non-equivalent method argument list on an object state. After we generate tests based on all combinations, we run all these generated tests and collect runtime information. From the collected runtime information, we extract the new non-equivalent object states that are encountered at runtime. We set them as the new frontier set. With this new frontier set, we start the subsequent iteration until we have reached the maximum iteration number or the frontier set has no object state. Then the algorithm returns the collected generated tests (in the form of JUnit [17] test classes) over all iterations.

For example, Parasoft Jtest 4.5 generates arguments for the `push` method of `StackWrapper` as -1, 0, and 7. Jtest also generates calls of `StackWrapper`'s constructor, `pop`, `increaseCount`, and `getAllStackCountPushCount` methods; these method calls do not have arguments. Then Rostra collects the constructor call into the set of non-equivalent constructor argument lists. Rostra also collects `push(-1)`, `push(0)`, `push(7)`, `pop()`, `increaseCount()`, and `getAllStackCountPushCount()` into the set of non-equivalent method argument lists. Before the first iteration, Rostra generates one test, which simply includes the constructor call. This constructor call produces an empty stack. Then in the first iteration, the following six tests are generated for `StackWrapper`, corresponding to the combinations of the empty stack and six non-equivalent method argument lists:

```
Test 1:
  StackWrapper s1 = new StackWrapper();
  s1.push(-1);

Test 2:
  StackWrapper s2 = new StackWrapper();
  s2.push(0);

Test 3:
  StackWrapper s3 = new StackWrapper();
  s3.push(7);

Test 4:
  StackWrapper s4 = new StackWrapper();
  s4.pop();

Test 5:
  StackWrapper s5 = new StackWrapper();
  s5.increaseCount();

Test 6:
  StackWrapper s6 = new StackWrapper();
  s6.getAllStackCountPushCount();
```

After we execute the tests generated during the first iteration, only 3 object states produced by Tests 1-3 are new. Then in the second iteration, we generate 18 tests, which correspond to the combinations of the 3 new object states and and 6 non-equivalent method argument lists. Below we show only one test for the combination of the new object state generated by Test 1 and the `push(-1)` non-equivalent method argument list:

```
Test 7:
  StackWrapper s7 = new StackWrapper();
  s7.push(-1);
  s7.push(-1);
```

4.4 Aspectual Coverage Measurement

In this section, we have measured the branch coverage within aspects. Because advice in aspects sometimes may not be woven into initially constructed base classes, some branches in the aspects might not be covered by the test inputs initially generated by test-generation tools (such as the ones described in preceding section). We additionally define and measure interaction coverage, whose measurement results can guide developers to improve base classes or test-generation tools, in order to cover those branches uncovered by the initially generated tests.

4.4.1 Aspectual Branch Coverage

A test adequacy criterion provides a stopping rule for testing and a measurement of test-suite quality [39]. A test adequacy criterion can be used to guide test selection. Because we do not have test oracles for those generated test inputs of AspectJ programs and it is not practical for developers to inspect a large number of generated tests, we can use test adequacy criteria to select generated test inputs for inspection. *Program-based criteria* [39] specify testing requirements based on whether all the identified features in a program have been fully exercised. Identified features in a program can be statements, branches, paths, or definition-use paths. In our research, we focus on whether all the identified branches in the aspects of an AspectJ program have been fully exercised. We call the coverage criteria *aspectual branch coverage*.

Aspectual branch coverage can be measured at the source code level or bytecode level. We decide to measure aspectual branch coverage at the bytecode level because the same piece of source code in aspects (e.g., source code in `around` advice) can be woven into multiple places in woven bytecode and covering these several places are often necessary for assuring high quality of the woven code. In principle, code coverage criteria defined at the bytecode level are stronger than the same ones defined at the source code level. In other words, if two test suites achieve the same coverage at the bytecode level, then these two test suites also achieve the same coverage at the source code level. However, if two test suites achieve the same coverage at the source code level, these two test suites may achieve different coverage at the bytecode level.

When measuring aspectual branch coverage at the bytecode level, we face several complications in tool implementation. First, we need to identify bytecode that is compiled from aspect source code. One possible solution is to develop our tool based on an AspectJ compiler such as ajc [1, 16] or abc [2, 7]. But the tool implementation based on an existing AspectJ compiler requires much development effort; in addition, much maintenance effort is needed to keep the tool implementation up to date when new versions of the compiler are released. The alternative solution, which we adopt, is to scan the woven bytecode based on some characteristics of the woven bytecode produced by an AspectJ compiler. In our tool implementation, we identify a class (in the bytecode form) produced by ajc [1, 16] to be an *aspect class* if it has a method whose name starts with "`ajc$`". The methods in an aspect class are *aspect methods*. Because some new methods of a base class are also created by ajc for advice such as `around` advice, we identify a method in a non-aspect class also to be an aspect method if its name ends with "`$advice`". Then the branches within an aspect method are instrumented automatically and their coverage is measured at runtime. Note that in this research context, a method entry is considered as one branch; therefore, measuring method coverage is part of measuring branch coverage.

The second complication is that some methods woven in the bytecode of an aspect class can be difficult or infeasible to be covered by tests generated for woven classes. For example, ajc creates a `hasAspect` method in the class bytecode of `NonNeagive` and no call site of this method is inserted in the base class. We prefer to leave this type of methods out of scope when we measure branch coverage. We have inspected uncovered methods and

branches measured by our tool and determine whether these uncovered methods or branches are infeasible or uninteresting to cover. If so, we improve our tool to exclude them from measurement.

4.4.2 Interaction Coverage

One goal of our research is to generate tests to cover all feasible branches in aspect methods. Often achieving this goal requires sufficient tool supports. The measurement results of aspectual branch coverage gives feedback to developers on what parts of aspect code are to be exercised. To give further guidance to developers on how to improve the aspectual branch coverage, we have defined and measured interaction coverage. The next section (Section 4.4.3) illustrates our methodology of using interaction coverage measurements in improving test generation. This interaction coverage criterion also indicates how well a test suite exercises the interactions among advised methods, advice, and intertype methods.

Our interaction coverage criterion is defined on the granularity of methods. Section 4.1 classifies four types of methods: advised methods, advice, intertype methods, and public non-advice methods. We call advice, intertype methods, and public non-advice methods as *aspect methods*. We particularly focus on the interactions between advised methods and aspect methods, and the interactions between aspect methods and aspect methods. We characterize interactions with method invocations. In particular, an interaction from method m_1 to method m_2 is characterized by a call site in m_1's body and this call site invokes m_2. We categorize interactions into the following three types:

- from advised methods to aspect methods (in short as *advised-aspect interaction*). One example advised-aspect interaction is the call site in `Stack`'s `insert` method body that invokes the `around` advice defined in `PushCount`.
- from aspect methods to aspect methods (in short as *aspect-aspect interaction*). One example aspect-aspect interaction is the call site in `PushCount`'s `around` advice body that invokes `PushCount`'s `increaseCount` method.
- from aspect methods to advised methods (in short as *aspect-advised interaction*). One example aspect-advised interaction is the call site in `NonNegative`'s `before` advice body that invokes `Stack`'s `iterator` method.

Note that we do not include interactions from advised methods to advised methods because we focus on testing aspectual behavior and aspectual composition behavior, rather than the interactions between advised methods in general.

An interaction is covered if its corresponding call site is covered. We measure interaction coverage as the number of covered interactions divided by all the identified interactions; in our tool's default configuration, we include only the advised-aspect and aspect-aspect interactions in the measurement because the coverage information of these two types of interactions can help us to improve aspectual branch coverage, as is presented in the next section.

4.4.3 Guidelines of Using Measurement Results

To increase aspectual branch coverage, developers can first focus on the coverage of aspect methods, because in order to cover branches within an aspect method, the aspect method needs to be covered first. We next illustrate how developers can use measurement results of interaction coverage to improve base-class construction and test generation.

Figure 6 shows the pseudo-code of using measurement results to improve base-class construction and test generation. Assume that an uncovered aspect method m is advice. If there exists no call site of it in base classes (that is, there is no interaction from any advised method to m), developers may improve base classes to make m to be woven into base classes. If there exists an uncovered call site of it in base classes (that is, there is an uncovered interaction from an advised method to m), developers may improve test generation such as adding relevant arguments to augment Jtest-generated arguments or increasing the user-defined maximum iteration number for Rostra.

```
void process(Set unCoveredAspectMethods) {
  foreach (m in unCoveredAspectMethods) {
    if (!isAdvice(m)) {
       Method n = getUpperMostNonPrivateCaller(m);
       if (n == null) {
           reportUnreachable();
           return;
       }
       if (isCovered(n)) {
           improveTestGen();
           return;
       }
       m = n;
    }
    Method l = getAdvisedMethCallerWithUnCovCallSite(m);
    if (l == null) {
        improveBaseClass();
    } else {
        improveTestGen();
    }
  }
}
```

Figure 6: Pseudo-code of using measurement results to improve base-class construction and test generation

Assume that an uncovered aspect method m is not advice. Normally m is not an intertype method or public non-advice method, because our wrapper mechanism assures that test-generation tools generate test inputs to cover them. Then m is likely to be a private non-advice method. We statically construct call chains for m with call site information that we collect during interaction-coverage instrumentation. If developers find out none of m's (either direct or indirect) callers is non-private (that is, none of them is advice, intertype method, or public non-advice method), m is inherently unreachable. If m's upper most non-private caller n is already covered, developers may improve test generation to generate test inputs to cover m through the call chain from n to m. If n is not covered, then we go through the preceding procedure for m when m is advice.

After going through the uncovered aspect methods, developers can focus on those uncovered branches within aspect methods. In order to cover these branches, developers may improve either base-class construction or test generation.

5. IMPLEMENTATION

Our implementation of Aspectra uses bytecode rewriting techniques based on the Byte Code Engineering Library (BCEL) [11] (instead of code instrumentation by using aspect-oriented paradigm). We have automated the wrapper synthesis by adapting a package in the Apache Avalon Framework [3] (also based on BCEL); this package generates wrapper classes for Java containers. Given the bytecode of a woven class as well as aspect classes, our tool automatically synthesizes a wrapper class in the form of bytecode.

We leverage Jtest [26] and our previously developed Rostra tool [32, 33] to generate test inputs. Rostra uses Java reflection mechanisms [5] to generate and execute new tests online. In the end of test generation, Rostra exports the test inputs generated after each iteration to a JUnit test class code [17]. More implementation de-

tails of Rostra can be found elsewhere [32, 33].

We have also automated the measurement of aspectual branch coverage and interaction coverage. Our tool reports the percentage numbers for branch coverage or interaction coverage. In addition, our tool reports the details of covered branches or call sites as well as uncovered branches or call sites. The details of a branch include the corresponding conditional, its line number in the source code, and the true or false branch of the conditional. The details of a call site include its line number in the source code and the corresponding caller and callee method names in the woven bytecode. To facilitate human inspection of these names, our future work plans to keep the mapping between a method name in the woven bytecode and the corresponding method name in the source code, and present to developers also the method name in the source code.

During class loading time, our tool dynamically determines whether a class is an aspect class by inspecting the names of its methods, because the ajc compiler [1, 16] gives special names for advice. We also similarly detect inlined `around` advice in base classes based on its method name. We scan bytecode to identify branches and then insert probes at branching points for collecting branch coverage information. We also scan the bytecode to identify call sites and classify them into different types of interactions, and then also insert probes for collecting interaction coverage information.

6. EXPERIENCE

This section presents our experience in applying Aspectra on 12 AspectJ benchmarks collected from a variety of sources (Section 6.1). We have applied Aspectra to generate test inputs for the collected benchmarks (Section 6.2). Our results suggest that our wrapper mechanism is necessary for testing several types of programs and our coverage measurement results are helpful for us to improve aspectual branch coverage (Section 6.3). We also discuss some issues of Aspectra (Section 6.4).

6.1 Benchmarks

Our benchmarks include most of the programs used by Rinard *et al.* [28] in evaluating their classification system for aspect-oriented programs. The benchmarks also include most of the programs[2] used by Dufour *et al.* [12] in measuring performance behavior of AspectJ programs. Our benchmarks also include one of the aspect-oriented design pattern implementations[3] by Hannemann and Kiczales [13].

Table 1 lists the benchmarks that we used. The first and second columns show the benchmark names and their advice/pointcut types, respectively. We measure the number of total branches in aspect code and the number of total call sites for the interaction coverage defined in Section 4.4.2, which are shown in the third and fourth columns, respectively. The aspect examples in Figure 3 are listed as the first three benchmarks, being `NonNegative`, `NonNegativeArg`, and `PushCount`. The `Instrumentation` benchmark is an aspect that counts the times of `push` calls. The `NullCheck` benchmark is an AspectJ program used by Asberry to detect whether method calls return null [6]. Following Rinard *et al.* [28], we refer to these 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 [20]. The `StateDesignPattern` benchmark had been implemented using AspectJ by Hannemann and Kiczales [13]. The `DCM` benchmark was implemented using AspectJ by Hassoun *et al.* [15] to validate their proposed dynamic coupling metric (DCM) [14]. The `ProdLine` benchmark was implemented using intertype declarations by Lopez-Herrejon and Batory for product lines of graph algorithms [24]. 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 [21]. 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.

[2]The AspectJ programs used by Dufour *et al.* [12] can be obtained from `http://www.sable.mcgill.ca/benchmarks/`.

[3]Hannemann and Kiczales's design pattern implementations can be obtained from `http://www.cs.ubc.ca/~jan/AODPs/`.

6.2 Procedures

In order to assess how our wrapper synthesis mechanism helps test generation, we first generate test inputs without using wrapper classes: we use the ajc compiler [1, 16] to weave aspects with base classes and then feed the resulting woven classes to Jtest 4.5 [26] to generate method arguments. Then Jtest-generated test classes are fed to our Rostra test-generation tool. We set Rostra's maximum iteration number as three. We measure the aspectual branch coverage and interaction coverage achieved by the generated tests. The measurement results are presented in Columns 5 and 6 of Table 1.

Next we repeat the preceding procedure except that we feed synthesized wrapper classes to Jtest and Rostra for test generation. Columns 7 and 8 list the measurement results of aspectual branch coverage and interaction coverage, respectively. We fill "–" in those entries that achieve the same measurement results as those produced without using wrapper classes (shown in Columns 5 and 6).

Finally, given the measurement results, according to the guidelines presented in Section 4.4.3, we improve either base class construction or test generation trying to get better coverage. Columns 9 and 10 list the measurement results of aspectual branch coverage and interaction coverage, respectively. Similarly we fill "–" in those entries that achieve the same measurement results as those produced by initially generated test inputs (shown in Columns 7 and 8).

6.3 Results

NonNegativeArg has an `execution` join point and using wrapper classes does not offer further help in test generation. The generated test inputs achieved 75% aspectual branch coverage. Our tool reported that the following call site is not covered: a call site (in `Stack`'s `iterator` method) that invokes the `before` advice in `NonNegativeArg`. We inspected the `Stack` code and found that `iterator` is not declared as `public` and Jtest or Rostra generates test inputs only for public methods. We then declared `iterator` as a public method and regenerated test inputs. The generated test inputs could achieve 100% interaction coverage. But our tool still reported that the false branch of `(args[i] instanceof Integer)` in `NonNegativeArg` (shown in Figure 3) was not covered. We inspected the `Stack` code and found that we needed a method that has at least one argument and this argument is not of the integer type. We added a public method `push(double d)` to the `Stack` class. Then generated inputs can achieve 100% aspectual branch coverage.

NonNegative has a `call` join point; therefore, without using wrapper classes, no aspectual branch coverage was achieved by generated test inputs (there were no interactions from the methods of the base class to the aspect methods). Then we fed the gener-

ated wrapper class to test-generation tools and the generated test inputs achieved 66% aspectual branch coverage and 100% interaction coverage. Our tool reported that the true branch of the following aspect code in NonNegative (shown in Figure 3) is not covered:

```
if (i < 0) throw new RuntimeException("negative");
```

We inspected the generated test code and found that test code contained method invocations of push(-1), which push negative elements into the stack. Then we further inspected the method body of push and found that the uncovered branch is due to the first line of push:

```
if (i < 0) return false;
```

After we commented out this line, which prevents negative elements from being finally pushed into the stack, the generated test inputs achieved 100% aspectual branch coverage.

PushCount has a public non-advice method getAllStackCount; therefore, without using wrapper classes, no coverage of getAllStackCount can be achieved by generated test inputs. After using the wrapper class, we achieve 100% aspectual branch coverage.

Instrumentation has two call join points and no aspectual branch coverage was achieved by generated test inputs for the woven class (without using a wrapper class). After we fed the generated wrapper class to test-generation tools, the generated test inputs achieved 100% aspectual branch coverage and 100% interaction coverage.

NullCheck has around advice for those methods whose returns are not void and not of primitive types:

```
Object around(): execution(Object+ *.*(..)) {
  Object lRetVal = proceed();
  if (lRetVal == null) {
    System.err.println(
    "Detected null return value after calling " +
    thisJoinPoint.getSignature().toShortString() +
    " in file " +
    thisJoinPoint.getSourceLocation().getFileName() +
    " at line " +
    thisJoinPoint.getSourceLocation().getLine());
 }
 return lRetVal;
}
```

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. Using wrapper classes does not offer further help in test generation. The generated test inputs achieved 25% aspectual branch coverage. Our tool reported that a call site (in Stack's iterator method) of the around advice in NullCheck is not covered. Similar to what we did for NonNegativeArg, we declared iterator as a public method and then the generated test inputs could achieve 100% interaction coverage and but 50% aspectual branch coverage. We found that the false branch of (lRetVal == null) is not covered, because return values of either pop or iterator can never be null. We modified Stack to be able to store any object type instead of just Integer and then pop can return an element that is null. Given the new base class, Jtest and Rostra generated test inputs that achieved 75% aspectual branch coverage. One remaining uncovered branch, the false branch of (lRetVal == null) in the inlined around advice for iterator, is infeasible to cover because iterator's return value can never be null.

Telecom has one key base class Connection. There are two aspects: Timing and Billing. The Timing aspect records the phone connection time and the Billing aspect uses the connection time to bill the dialer. Either Timing or Billing aspect declares two pieces of after advice with call join points. Both aspects also declare intertype fields and methods. Without using a wrapper class, branches within advice with call join points cannot be covered. In addition, because Timing and Billing define three public non-advice methods, before using wrapper classes, these three methods were not covered. After using wrapper classes, generated test inputs achieved 100% aspectual branch coverage and 100% interaction coverage.

BusinessRuleImpl has a base class of SavingsAccount and two aspects: MinimumBalanceRuleAspect and OverdraftProtectionRuleAspect. MinimumBalanceRuleAspect defines a piece of before advice for method execution and OverdraftProtectionRuleAspect defines another piece of before advice for method execution. Our tool reported that generated test inputs achieved only 50% aspectual branch coverage. We inspected those uncovered branches and found that some arguments generated by Jtest are not sufficient. For example, MinimumBalanceRuleAspect required the minimum balance to be 25 but the Jtest-generated arguments for SavingsAccount's credit method are only -1, 0, or 7; after they are invoked even for three iterations, SavingsAccount still could not get sufficient funds for withdrawal. We improved Jtest-generated method arguments by adding some new arguments to improve the aspectual branch coverage to 80% but we could not easily improve test generation to exercise two uncovered branches because the complexity of necessary conditions for covering these two branches is beyond our capability. A more sophisticated test-generation tool is needed for generating tests to cover them.

StateDesignPattern has a QueueStateAspect aspect that declares three pieces of after advice for method calls. Because these advised call sites already exist in the code base, using wrapper classes does not offer further help in test generation. Our tool reported that generated test inputs achieved 88% aspectual branch coverage, with one branch uncovered. We inspected the uncovered branch, which indicated that the base class Queue has not reached the full state yet. We therefore increased Rostra's maximum iteration number and when the number was four, we achieved 100% aspectual branch coverage.

DCM has an Metrics aspect that uses around and after advice for method executions. Our tool reported 34% aspectual branch coverage. We found that seven aspect methods are uncovered, two of which are uncovered advice methods. By inspecting these advice method's join point definitions, we found they advise main method in the base class but our base class did not have a main method. We constructed a main method for the base class and then six of these seven originally uncovered methods were covered. The aspectual branch coverage was increased from 34% to 52%. By inspecting the remaining uncovered branches, many of them were due to that the value of Metrics's static field dumpinterval was never 1 but its value is preset as -1 and could not be updated through code interface (that is, these branches cannot be covered inherently unless we modify the code under test). For other uncovered branches, we could not easily improve test generation to cover them. A more sophisticated test-generation tool is needed for generating tests to cover them.

ProdLine has base classes that 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. Note that our coverage measurement tool measures all aspect classes loaded at class loading time. Many other loaded aspect classes than DFS and Undirected were loaded and thus measured. Our tool reported low aspectual branch coverage (28%) and interaction coverage (13%). We inspected these uncovered branches or call sites and found that many of them

AspectJ program	advice type: pointcut type	branches	callsites	before wrapping		after wrapping		after guidance	
				%branch	%inter	%branch	%inter	%branch	%inter
NonNegativeArg	before:exec	4	4	75%	50%	–	–	100%	100%
NonNegative	before:call	3	3	0%	0%	66%	100%	100%	–
PushCount	around:exec;inter	4	3	75%	100%	100%	–	–	–
Instrumentation	after:call	2	1	0%	0%	100%	100%	–	–
NullCheck	around:exec	4	2	25%	50%	–	–	75%	100%
Telecom	after:call	14	10	85%	70%	100%	100%	–	–
BusinessRuleImpl	before:exec	10	7	50%	100%	–	–	80%	–
StateDesignPattern	after:call	9	2	88%	100%	–	–	100%	–
DCM	around/after:exec	50	16	34%	38%	–	–	52%	87%
ProdLine	inter	141	74	28%	13%	–	–	–	–
Bean	around:exec;inter	10	10	100%	100%	–	–	–	–
LoD	before:call/exec	22	70	45%	24%	59%	67%	68%	80%

Table 1: Results of applying Aspectra to generate test inputs for 11 subjects

are infeasible to be covered by those test inputs generated for `Vertex`. To achieve high coverage of all aspects, we would need to generate test inputs for all these base classes. When we focused on the coverage of `DFS` and `Undirected`, our generated test inputs got reasonably sufficient coverage.

Bean has a base class `Point`. The `BoundPoint` aspect declares five intertype methods. Then the aspect uses `around` advice for method execution. Without using wrapper classes, Jtest and Rostra generated test inputs that achieved 100% aspectual branch coverage and 100% interaction coverage.

LoD has a `Check` aspect that 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 `percflow`, `pertarget`, and `cflow`. Our tool reported that generated test inputs achieved 45% aspectual branch coverage. After we used a wrapper class, the aspectual branch coverage was increased to 59%; the increase is due to the increase of call depths affecting LoD's behavior, rather than our original intention of the wrapper mechanism. We further inspected uncovered advice and found a piece of uncovered advice advises a `main` method and another piece advises Java library method calls. We constructed a `main` method in the base class and put some call sites of Java library methods in the `main` method. Then the generated test inputs achieved 68% aspectual branch coverage. We could not easily improve base-class construction or test generation for the remaining uncovered branches. Besides a more sophisticated test-generation tool, a sophisticated base-class construction tool is needed for helping further improve the aspectual branch coverage.

6.4 Discussion

Although we applied Aspectra on only 12 AspectJ benchmarks, we expect that Aspectra can be applied to a wide range of AspectJ programs. In our experience, the wrapper mechanism is useful in test generation when advice is `call` advice and there are no call sites of its advised methods in the code base. In addition, the wrapper mechanism is useful in test generation for public non-advice methods of aspect classes. From our experience with testing these 12 benchmarks, constructing appropriate base classes turned out to be more important than we originally thought, especially for complex aspects. As we discussed in Section 6.1, we simply used `Stack` for those aspects that were not equipped with base classes. It turned out that `Stack` may not be sufficient for some aspects such as `DCM` and `LoD`. The measurement results of both interaction coverage and aspectual branch coverage are helpful for guiding us in improving base classes as well as test generation. But at the same time, we also noticed that it is sometimes not trivial to manually figure out how to improve test coverage, especially when we were not familiar with the code base under test. This observation suggests the need of developing tools to automatically construct appropriate base classes given aspects. We plan to pursue this research direction in our future work.

The runtime overhead of applying Aspectra is low. Because our coverage measurement is focused on aspects or interactions between aspects and base classes, the runtime overhead of measurement is negligible. Although running Rostra with a large number of iterations could be expensive, all 12 benchmarks except for `StateDesignPattern` require no more than three iterations to achieve optimal aspectual branch coverage with Jtest-generated arguments.

Sometimes Jtest-generated arguments are not sufficient (such as those for `BusinessRuleImpl`). We expect that using a more powerful tool with symbolic execution [31, 34] could generate better arguments to cover some branches and thus reduce manual efforts in producing relevant method arguments based on coverage measurement results.

Aspectra focuses on testing aspectual behavior. Although Aspectra also measures interaction coverage and tries to cover the interactions between aspects and base classes, testing aspectual composition behavior may require developments of new techniques.

Aspectra tests aspectual behavior of an aspect by weaving the aspect with one or several constructed base classes (including only several representative base classes if the total number of base classes is too large). Then it leverages the existing test-generation tools to generate test inputs on a per-class basis. Indeed, in the context of AOP, an aspect typically impacts more than one class and the aspect is usually executed in a broader context than what Aspectra has tested. However, as was argued by Lopes and Ngo [23], testing aspectual behavior individually can be cost-effective in diagnosing failures and detecting faults in aspectual implementations. In addition, they found that the majority of existing aspects are general purpose such as logging, tracing, persistence, profiling, and design patterns. Testing these aspects woven with some representative base classes can often be effective in detecting faults in aspectual implementations before integration testing is done.

7. RELATED WORK

Souter *et al.* [29] developed a test selection technique based on concerns. A concern is the code associated with a particular maintenance task. An aspect in AspectJ programs can be seen as a concern. To reduce the space and time cost of running tests on instrumented code, they proposed to instrument only the concerns for collecting runtime information. They also proposed to select or

prioritize tests for the selected concerns. In particular, they select a test if the test covers a concern that has not been been exercised by previously selected tests. Zhou *et al.* [38] also used the same technique for selecting tests for an aspect. These two test selection approaches assume that there already exist a set of tests for an AspectJ program (or just for the base classes in the AspectJ program), whereas Aspectra focuses on automatically generating test inputs for an AspectJ program and using coverage measurement results to guide developers to improve test coverage. Our aspectual branch coverage or interaction coverage can also be used to select tests: if a test input covers at least one new aspectual branch or interaction, the test input is selected for inspection when there are no test oracles for generated test inputs. Our aspectual branch coverage or interaction coverage defines coverage in the granularity of branches or call sites, whereas Souter *et al.* or Zhou *et al.* define coverage in a coarser granularity of whole aspects.

Xu *et al.* [35] presented a specification-based testing approach for aspect-oriented programs. The approach creates aspectual state models by extending the existing FREE (Flattened Regular Expression) state model, which was originally proposed for testing object-oriented programs. Based on the model, they developed two techniques for testing aspect-oriented programs. The first technique transforms an aspectual state model to a transition tree and generates tests based on the tree. The second technique constructs and searches an aspect flow graph for achieving statement coverage and branch coverage. Their work focuses on testing aspect-oriented programs based on abstract state models, whereas Aspectra focuses mainly on automatically generating test inputs based on implementations.

Alexander *et al.* [4] developed a fault model for aspect-oriented programming, including six types of faults that may occur in aspect-oriented systems. Their fault model provides useful guidance in developing testing coverage tools for aspect-oriented programs, whereas Aspectra proposes an automated approach for generating tests to achieve structural coverage. Recently, Nathan and Alexander [25] proposed another fault model that results from maintenance problems for aspect-oriented programs. They also reported their evaluation of AspectJ's ability to distribute class files containing woven concerns and to reweave them later. Although their fault model may lead to a better understanding of maintenance issues inherent in aspect-oriented software, it still does not provide a testing solution for aspect-oriented programs.

Zhao [36, 37] 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 automatically generating test inputs for AspectJ programs.

Rajan and Sullivan [27] 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 generating test inputs for AspectJ programs and using structural coverage measurement results to guide how to improve test coverage.

Rinard *et al.* [28] proposed a classification system for aspect-oriented programs and developed a static analysis to support automatic classification. Their system characterizes the interactions between advice and advised methods based on field accesses. Developers can use the classification system and analysis to structure their understanding of the aspect-oriented programs. Aspectra defines and measures the coverage of interactions between advised methods and aspect methods in the granularity of methods.

8. CONCLUSION

We proposed Aspectra, a novel framework for automatically generating test inputs for AspectJ programs. To test aspects in an AspectJ program, developers can construct base classes, which can be woven with aspects to produce woven classes. Aspectra synthesizes a wrapper class for each woven class. The wrapper mechanism allows test-generation tools to indirectly exercise advice related to `call` join points and public non-advice methods in aspects during test generation. At the same time, the mechanism prevents the methods in generated test classes from being advised by unwanted advice. Given wrapper classes, Aspectra leverages existing test-generation tools for generating test inputs. But sometimes behavior in aspects may not be sufficiently exercised by test inputs initially generated by these tools for base classes constructed by developers. To assess the quality of generated test inputs, we define and measure aspectual branch coverage and interaction coverage based on four types of methods in AspectJ programs: advised methods, advice, intertype methods, and public non-advice methods. We provide guidelines for developers to use measurement results to improve base-class construction or test generation. Our experience shows that our wrapper mechanism is necessary for some important types of AspectJ programs and our measurement results provide useful guidance for improving test coverage.

In future work, we plan to adapt our framework implementation to accommodate AspectJ code compiled by other compilers such as abc [2, 7]. We plan to provide tool supports for automatic construction of base classes given aspects. We also plan to extend our framework to address testing of other important behavior such as aspectual composition behavior, including not only interactions between base classes and aspects but also interactions among multiple aspects.

Acknowledgments

We thank Darko Marinov for discussion on this work and comments on a previous draft of this paper. We thank Parasoft Co. for providing the Jtest tool to us.

9. REFERENCES

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

[2] abc: The AspectBench Compiler for AspectJ, version 1.0.2, Feburary 2005. `http://aspectbench.org/`.

[3] Apache Avalon Framework, August 2005. `http://excalibur.apache.org/`.

[4] 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.

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

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

[7] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni,

G. Sittampalam, and J. Tibble. abc: an extensible AspectJ compiler. In *Proc. 4th International Conference on Aspect-Oriented Software Development*, pages 87–98, 2005.

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

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

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

[11] M. Dahm and J. van Zyl. Byte Code Engineering Library, April 2003. `http://jakarta.apache.org/bcel/`.

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

[13] 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.

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

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

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

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

[18] 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.

[19] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

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

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

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

[23] C. V. Lopes and T. Ngo. Unit testing aspectual behavior. In *Proc. AOSD 05 Workshop on Testing Aspect-Oriented Programs*, March 2005.

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

[25] N. McEachen and R. Alexander. Distributing classes with woven concerns - a look into potential fault scenarios. In *Proc. 4th International Conference on Aspect-Oriented Software Development*, pages 192–200, March 2005.

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

[27] H. Rajan and K. Sullivan. Aspect language features for concern coverage profiling. In *Proc. 4th International Conference on Aspect-Oriented Software Development*, pages 181–191, March 2005.

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

[29] A. L. Souter, D. Shepherd, and L. L. Pollock. Testing with respect to concerns. In *Proc. International Conference on Software Maintenance*, page 54, 2003.

[30] 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.

[31] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *Proc. 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 97–107, 2004.

[32] 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.

[33] 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.

[34] T. Xie, D. Marinov, W. Schulte, and D. Noktin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proc. the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, April 2005.

[35] D. Xu, W. Xu, and K. Nygard. A state-based approach to testing aspect-oriented programs. In *Proc. 17th International Conference on Software Engineering and Knowledge Engineering*, July 2005.

[36] 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.

[37] 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.

[38] 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.

[39] 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.