Data-Flow-Based Unit Testing of Aspect-Oriented Programs

Jianjun Zhao
Department of Computer Science and Engineering
Fukuoka Institute of Technology
3-30-1 Wajiro-Higashi, Higashi-ku, Fukuoka 811-0295, Japan
zhao@cs.fit.ac.jp

Abstract

The current research so far in aspect-oriented software development is focused on problem analysis, software design, and implementation techniques. Even though the importance of software testing is known, it has received little attention in the aspect-oriented paradigm. In this paper, we propose a data-flow-based unit testing approach for aspect-oriented programs. Our approach tests two types of units for an aspect-oriented program, i.e., aspects *that are modular units of crosscutting implementation of the program, and those* classes *whose behavior may be affected by one or more aspects. For each aspect or class, our approach performs three levels of testing, i.e.,* intra-module, inter-module, *and* intra-aspect *or* intra-class *testing. For an individual module such as a piece of advice, a piece of introduction, and a method, we perform intra-module testing. For a public module along with other modules it calls in an aspect or class, we perform inter-module testing. For modules that can be accessed outside the aspect or class and can be invoked in any order by users of the aspect or class, we perform intra-aspect or intra-class testing. Our approach can handle unit testing problems that are unique to aspect-oriented programs. We use control flow graphs to compute def-use pairs of an aspect or class being tested and use such information to guide the selection of tests for the aspect or class.*

1 Introduction

Aspect-oriented software development (AOSD) is a new technique to support separation of concerns in software development [3, 9, 13, 16]. The techniques of AOSD make it possible to modularize crosscutting aspects of a system. Like objects in object-oriented software development, aspects in AOSD may arise at any stage of the software life cycle, including requirements specification, design, implementation, etc. Some examples of crosscutting aspects are exception handling, synchronization, and resource sharing.

The current research so far in AOSD is focused on problem analysis, software design, and implementation techniques. Even though the importance of software testing and verification is known, it has received little attention in the aspect-oriented paradigm. Although it has been claimed that applying an AOSD method will eventually lead to quality software, aspect-orientation does not provide 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 are by no means shields against programmer's mistakes or a lack of understanding of the specification. As a result, software testing remains an important task even in AOSD.

Aspect-oriented programming introduces some new language constructs such as join points, advice, introduction, aspects, that differ from procedural and object-oriented programs. These specific constructs in aspect-oriented programs require special testing support and provide opportunities for exploitation by a testing strategy. However, although many testing approaches have been proposed for procedural and object-oriented programs, there is no testing approach for aspect-oriented programs until now. Also, the existing testing approaches for procedural and object-oriented programs can not be applied directly to aspect-oriented programs. Therefore, new testing techniques and tools that are appropriate for testing aspect-oriented programs are needed.

Unit testing is to test each unit (basic component) of a program to verify that the detailed design for the unit has been correctly implemented [22]. Since unit testing is performed after implementing a program's unit (component), it is very effective to check various errors in a program's units at an earlier stage of its life cycle. There are two types of unit testing, i.e., *specification-based* unit testing (*black-box testing*), and *program-based* unit testing (*white-box testing*). Whereas specification-based testing focuses on verifying the functions and behaviors of software components according to an external view, program-based testing focuses on checking the internal logic structures and behaviors of a software component. One type of program-based testing is *data flow testing* [8, 12, 19] which tests how values which are associated with variables can affect the execution of the program. Data flow testing uses the data flow relations in a program to guide the selection of tests.

In aspect-oriented programs, the basic testing unit is an aspect (or class). An aspect (or class) is designed to work as independently as possible from its environment. This is a benefit to unit testing, since it allows the programmer to write a small testing program to exercise the aspect (or class) alone. However, on the other hand, an aspect may affect the behavior of one or more classes through advice and introduction, making the interactions between the aspect and affected classes more complex. Therefore, when performing unit testing on an aspect or class, one should consider not only the aspect or class being tested but also those classes whose behavior may be affected by the aspect being tested and those aspects that may affect the behavior of the class being tested.

In this paper, we propose a data-flow-based unit testing approach by combining the unit testing and data flow testing techniques to test aspects and classes in an aspect-oriented program. By supporting data flow testing of aspects or classes, our approach provides opportunities to find errors in aspects or classes that may not be covered by using specification-based testing.

Our unit testing approach tests two types of units for an aspect-oriented program, i.e., *aspects* that are modular units of crosscutting implementation of the program, and those *classes* whose behavior may be affected by one or more aspects. For each aspect or class, our approach performs three levels of testing, i.e., *intra-module*, *inter-module*, and *intra-aspect* or *intra-class* testing. For an individual module such as a piece of advice, a piece of introduction, and a method, we perform intra-module testing. For a public module along with other modules it calls in an aspect or class, we perform inter-module testing. For modules that can be accessed outside the aspect or class and can be invoked in any order by users of the aspect or class, we perform intra-aspect or intra-class testing. Our approach can handle unit testing problems that are unique to aspect-oriented programs. We use control flow graph to compute def-use pairs of an aspect or class being tested and use such information to guide the selection of tests for the aspect or class.

The rest of the paper is organized as follows. Section 2 briefly introduces the AspectJ and data flow analysis and testing. Section 3 presents a motivating example to discuss some issues that arise in testing aspects or classes. Section Section 4 presents a structure model for unit testing of aspect-oriented programs. 5 describes a data-flow-based unit testing approach for aspect-oriented programs. Section 6 briefly introduces the tool support for our testing approach. Section 7 discusses related work. Concluding remarks are given in Section 8.

2 Background

2.1 Aspect-Oriented Programming with AspectJ

We present our data-flow-based unit testing approach of aspect-oriented programs in the context of AspectJ, the most widely used aspect-oriented programming language [10]. Our basic techniques, however, deal with the basic concepts of aspect-oriented programming and therefore apply to the general class of aspect-oriented languages.

AspectJ [10] is a seamless aspect-oriented extension to Java; AspectJ adds some new concepts and associated constructs to Java. These concepts and associated constructs are called join point, pointcut, advice, introduction, and aspect. We briefly introduce each of these constructs as follows.

The *aspect* is the 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 also may be specialized with sub-aspects. An aspect is combined with the classes it crosscuts according to specifications given within the aspect. Moreover, an aspect can use an *introduction* construct to introduce methods, attributes, and interface implementation declarations into classes. Introduced members may be made visible to all classes and aspects (public introduction) or only within the aspect (private introduction), allowing one to avoid name conflicts with pre-existing elements. For example, the aspect `PointShadowProtocol` in Figure 1 privately introduces a field `shadow` to the class `Point` at `s31`.

A central concept in the composition of an aspect with other classes is called 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, an exception handler, etc. Sets of join points may be represented by *pointcuts*, implying that such sets may crosscut the system. Pointcuts can be composed and new pointcut designators can be defined according to these combinations. AspectJ provides various pointcut *designators* that may be combined through logical operators to build up complete descriptions of pointcuts of interest. For example, the aspect `PointShadowProtocol` in Figure 1 declares three pointcuts named `setting`, `settingX`, and `settingY` at `p36`, `p37`, and `p38`.

An aspect can specify *advice*, which is used to define code that executes when a pointcut is reached. Advice is a method-like mechanism which consists of instructions that execute *before*, *after*, or *around* a pointcut. `around` advice executes *in place* of the indicated pointcut, allowing a method to be replaced. For example, the aspect `PointShadowProtocol` in Figure 1 declares three pieces of after advice at `ae39`, `ae43`, and `ae48`; each is attached to the corresponding pointcut `setting`, `settingX`, or `settingY`.

An AspectJ program can be divided into two parts: *base*

```
 ce0 public class Point {
  s1   protected int x, y;
 me2   public Point(int _x, int _y) {
  s3     x = _x;
  s4     y = _y;
       }
 me5   public int getX() {
  s6     return x;
       }
 me7   public int getY() {
  s8     return y;
       }
 me9   public void setX(int _x) {
 s10     x = _x;
       }
me11   public void setY(int _y) {
 s12     y = _y;
       }
me13   public void printPosition() {
 s14     System.out.println("Point at("+x+","+y+")");
       }
me15   public static void main(String[] args) {
 s16     Point p = new Point(1,1);
 s17     p.setX(2);
 s18     p.setY(2);
       }
     }
ce19 class Shadow {
 s20   public static final int offset = 10;
 s21   public int x, y;

me22   Shadow(int x, int y) {
 s23     this.x = x;
 s24     this.y = y;
me25   public void printPosition() {
 s26     System.outprintln("Shadow at
                ("+x+","+y+")");
       }
     }
```

```
ase27 aspect PointShadowProtocol {
  s28   private int shadowCount = 0;
 me29   public static int getShadowCount() {
  s30     return PointShadowProtocol.
                aspectOf().shadowCount;
        }
  s31   private Shadow Point.shadow;
 me32   public static void associate(Point p, Shadow s){
  s33     p.shadow = s;
        }
 me34   public static Shadow getShadow(Point p) {
  s35     return p.shadow;
        }

 pe36   pointcut setting(int x, int y, Point p):
          args(x,y) && call(Point.new(int,int));
 pe37   pointcut settingX(Point p):
          target(p) && call(void Point.setX(int));
 pe38   pointcut settingY(Point p):
          target(p) && call(void Point.setY(int));

 ae39   after(int x, int y, Point p) returning :
            setting(x, y, p) {
  s40     Shadow s = new Shadow(x,y);
  s41     associate(p,s);
  s42     shadowCount++;
        }
 ae43   after(Point p): settingX(p) {
  s44     Shadow s = new getShadow(p);
  s45     s.x = p.getX() + Shadow.offset;
  s46     p.printPosition();
  s47     s.printPosition();
        }
 ae48   after(Point p): settingY(p) {
  s49     Shadow s = new getShadow(p);
  s50     s.y = p.getY() + Shadow.offset;
  s51     p.printPosition();
  s52     s.printPosition();
       }
     }
```

Figure 1: A sample AspectJ program.

code which includes classes, interfaces, and other standard Java constructs and *aspect code* which implements the crosscutting concerns in the program. For example, Figure 1 shows an AspectJ program that associates shadow points with every `Point` object. The program can be divided into the base code containing the classes `Point` and `Shadow`, and the aspect code which has the aspect `PointShadowProtocol` that stores a shadow object in every `Point`. Moreover, the AspectJ implementation ensures that the aspect and base code run together in a properly coordinated fashion. The key component is the *aspect weaver*, when ensures that applicable advice runs at the appropriate join points. For more information about AspectJ, refer to [2].

2.2 Data Flow Testing

Data flow testing [8, 12, 19] mainly focuses on testing the value assignment of each variable in a program by executing sub-paths from the assignment (*definition*) to some program points in which the variable is used (*use*). The use of a variable can be classified as either a computation use or a predicate use. If the value of a variable is used in computing a value for defining other variables or as an output value in an output statement, the use is called *computation use*, denoted as *c-use*. Otherwise, if the value of a variable is used to decide the result of a predicate statement for selecting execution paths, the use is called *predicate use*, denoted as *p-use*. A *def-use pair* of a variable v is an order pair (d, u) such that d is a statement having a definition of v and u is a statement having a use of v, or some memory location bound to v, that can be reached by d over some path in the program.

Data flow testing criteria are used to select particular def-use associations to test. A test satisfies a def-use pair if the execution of the program with the test lead to traversal of a subpath from the definition to the use without any interventing redefinition of that memory location [8]. For c-uses, tests must travel from the statement containing the definition to the statement containing the compution use. For p-uses, tests must travel from a statment containing the definition to both executional successors of the statement containing the predicate use. Many types of data flow testing criteria have been defined [6, 12]. Among them, the "all-definitions" criterion requires that a test should cover a path from each definition to a use, while the "all-uses" criterion requires that one subpath from every definition to each of its uses is tested.

3 Motivating Example

We present a motivating example to discuss the issues that arise in testing aspects and classes of aspect-oriented programs.

Consider the aspect `PointShadowProtocol` shown in Figure 1, that declares a piece of after advice (with the `settingX` pointcut) to modify the behavior of class `Point`. This after advice can be applied to each join point where a target object of type `Point` receives a call to the method with signature `Point.setX(int)`. The `target` keyword is used to give the name to the target object.

In AspectJ this after-advice is applied by the compiler without explicit reference to the aspect from the `Point` class. So when testing class `Point`, by definition, existing unit testing approaches only test the `Point` class itself, but does not consider the effect from the `PointShadowProtocol` aspect. However, when class `Point` and aspect `PointShadowProtocol` are compiled together, then intuitively the behavior of `Point`'s `setX` method may be changed due to the after-advice. As a result, in order to correctly testing class `Point`, we should take into account not only the `Point` class itself but also the `PointShadowProtocol` aspect that may affect the behavior of `Point` through the after-advice. On the other hand, consider that we want to perform testing on aspect `PointShadowProtocol`. We should not just test the aspect itself, because the after-advice in the aspect just specifies a partial behavior of the methods declared in class `Point`, and also because the after-advice is automatically woven into some methods in the `Point` class by the compiler, and therefore no call exists for the after-advice. So when we perform testing on aspect `PointShadowProtocol`, we should test the `PointShadowProtocol` together with those methods in class `Point`, whose behavior may be affected by the advice from the `PointShadowProtocol`. Unfortunately, existing unit testing approaches can handle neither of these cases.

As a result, it is impractical to test an aspect or class in isolation in an aspect-oriented program. To correctly test aspects and classes, we should (1) test an aspect together with those methods whose behavior may be affected by the aspect's advice (*from the aspect perspective*), and (2) test a class together with those pieces of advice that may affect its behavior and those pieces of introduction that may introduce some new members to the class (*from the class perspective*).

To make this possible, we define some necessary notions below.

- A *clustering aspect*, denoted by *c-aspect*, is an individual aspect together with some methods in one or more classes, such that the methods' behavior may be affected by the aspect's advice.

- A *clustering class*, denoted by *c-class*, is an individual class together with some pieces of advice and introduction in one or more aspects, such that the advice may affect the behavior of the class's methods, and the introduction may change the type structure of the class.

- A *clustering method*[1], denoted by *c-method*, is an individual method together with one or more pieces of advice that may affect the method's behavior.

- A *normal class*, denoted by *n-class*, is an individual class whose behavior may never be affected by some aspect.

- A *normal method*, denoted by *n-method* is an individual method whose behavior will never be affected by some piece of advice.

Based on the above definitions, in this paper we focus on testing c-aspect and c-class units in an aspect-oriented program; we do not consider n-classes of the program because these n-classes can be tested, for example, by using existing class testing techniques proposed in [4, 8].

Example. Consider the AspectJ program shown in Figure 1. From the above definitions, the c-aspect of aspect `PointShadowProtocol`, which is also called `PointShadowProtocol`, should contain the aspect itself as well as methods `Point`, `setX`, and `setY` in class `Point`. The c-class of class `Point`, which is also called `Point`, should contain the class itself as well as those pieces of after-advice associated with pointcuts `setting`, `settingX`, and `settingY` respectively, because the behavior of its constructor `Point` and two methods `setX` and `setY` may be changed by the after-advice declared in `PointShadowProtocol`. On the other hand, `Shadow` is a normal class since no aspect exists that may affect its behavior.

In the rest of the paper, we use the same name to represent each c-aspect, c-class, n-aspect, or n-class and its original aspect or class, and also the same name to represent each c-method or n-method and its original method. Moreover, to keep our terminology consistent in the rest of paper, we use the word "module" to stand for a c-method, a piece of introduction, or a n-method in a c-aspect or c-class.

4 The Structure Model for Unit Testing of Aspect-Oriented Programs

To facilitate data flow testing of c-aspects and c-classes in aspect-oriented programs, we propose a structure model that captures the data flow test artifacts of a c-aspect or c-class. From this model, data flow test cases can be derived

[1]We treat a constructor in a class as a special case of a method, and similar to the clustering method, we can define a *clustering constructor*, denoted by *c-constructor*, and a *normal constructor*, denoted by *n-constructor*.

```
algorithm BuildACFG
  input A c-aspect $A$
  output The framed Control Flow Graph (FCFG) $G_{acfg}$ of $A$
  declare
  begin BuildACFG
    /* Build the call graph for $A$ and add to $G_{acfg}$ */
        $G_{acfg}$ = Construct the call graph for $A$
    /* Add the frame to the call graph and add to $G_{acfg}$ */
        $G_{acfg} = G_{acfg} \cup frame$
    /* Build CFGs for c-methods, introduction, and n-methods in $A$
                  and add to $G_{acfg}$ */
    /* Replace each call graph vertex with the corresponding CFG */
    for each c-method, introduction, or n-method $\alpha$ in $A$ do
      Replace $\alpha$'s call graph vertex in $G_{acfg}$ with $\alpha$'s CFG
      Update arcs appropriately
    endfor
    /* Replace call sites with call and return vertices */
    for each call vertex $s$ in $G_{acfg}$, representing a call
            to c-method, introduction, or n-method $\alpha$ in $A$ do
      Replace $s$ with a call and a return vertex
      Update arcs appropriately
    endfor
    /* Connect the individual CFGs */
    for each c-method, introduction, or n-method $\alpha$ in $A$ do
      Add an arc from the frame call vertex to the start
      vertex of $\alpha$'s CFG in $G_{acfg}$
      Add an arc from the exit vertex of $\alpha$'s CFG in $G_{acfg}$
      to the frame return vertex
    endfor
    /* Return the complete FCFG of $A$ */
        return $G_{acfg}$
  end BuildACFG
```

Figure 2: An Algorithm for constructing the FCFG of a c-aspect.

to ensure the quality of the c-aspect or c-class. The structure model consists of three different types of control flow graphs in order to capture different levels of data flow information in a c-aspect or c-class. We present each type of the graphs as follows.

4.1 Control Flow Graphs for Individual Modules

We use the *control flow graph* (CFG) as a basis to obtain the data flow information in a module (i.e., a c-method, a n-method, or a piece of introduction) of a c-aspect or c-class.

The *control flow graph* (CFG) of a module represents the control flow relationships that exist within the module, and contains a vertex for each statement or predicate expression in the module and arcs between vertices that represent flow of control between statements. There is a unique vertex called *entry vertex* to represent the unique entry to the module, and a unique vertex called *exit vertex* to represent exit from the module.

To represent the data flow information in a module, the CFG of the module is annotated by the definition/use of the variables so that the def-use pairs for the variables of interest can be obtained from the CFG.

```
algorithm BuildCCFG
  input A c-class $C$
  output The Framed Control Flow Graph (FCFG) $G_{ccfg}$ of $C$
  declare
  begin BuildCCFG
    /* Build the call graph for $C$ and add to $G_{ccfg}$ */
    $G_{ccfg}$ = Construct the call graph for $C$
    /* Add the frame to the call graph and add to $G_{ccfg}$ */
    $G_{ccfg} = G_{ccfg} \cup frame$
    /* Build CFGs for c-methods, n-methods in $C$ and add to $G_{ccfg}$ */
    /* Replace each call graph vertex with the corresponding CFG */
    for each c-method, n-method $m$ in $C$ do
      Replace $m$'s class call graph vertex in $G_{ccfg}$ with $m$'s CFG
      Update arcs appropriately
    endfor
    /* Replace call sites with call and return vertices */
    for each call vertex $s$ in $G_{ccfg}$, representing a call
            to c-method or n-method $m$ in $C$ do
      Replace $s$ with a call and a return vertex
      Update arcs appropriately
    endfor
    /* Connect the individual CFGs */
    for each c-method or n-method $m$ in $C$ do
      Add an arc from the frame call vertex to the start
      vertex of $m$'s CFG in $G_{ccfg}$
      Add an arc from the exit vertex of $m$'s CFG in $G_{ccfg}$
      to the frame return vertex
    endfor
    /* Return the complete FCFG of $C$ */
        return $G_{ccfg}$
  end BuildCCFG
```

Figure 3: An Algorithm for constructing the FCFG of a c-class.

4.2 Interprocedural Control Flow Graphs for Interactive Modules

We use the *interprocedural control flow graph* (ICFG) as a basis to obtain the data flow information that involves more than one module in a c-aspect or c-class. The ICFG represents the control flow relationships that exist within and among a group of modules. The ICFG is composed of a call graph which is a directed graph whose vertices represent modules and arcs represent the possible calls between modules, and a group of CFGs each of which represents a module in the c-aspect or c-class. Since the ICFG allows testers to integrate the CFGs of calling related modules into a single entry, single exit CFG, one can obtain the def-use pairs for a variable that is defined in one module and used in other modules based on the ICFG. The ICFG can be constructed using the algorithm developed in [17].

4.3 Framed Control Flow Graphs for Aspects and Classes

We use the *framed control flow graph* (FCFG) as a basis to derive the data flow information in a c-aspect or c-class. The FCFG consists of a collection of CFGs each of which presents a module in the c-aspect or c-class and some additional arcs used to construct the frame. In an FCFG, there are some vertices used to represent the frame such as *frame*

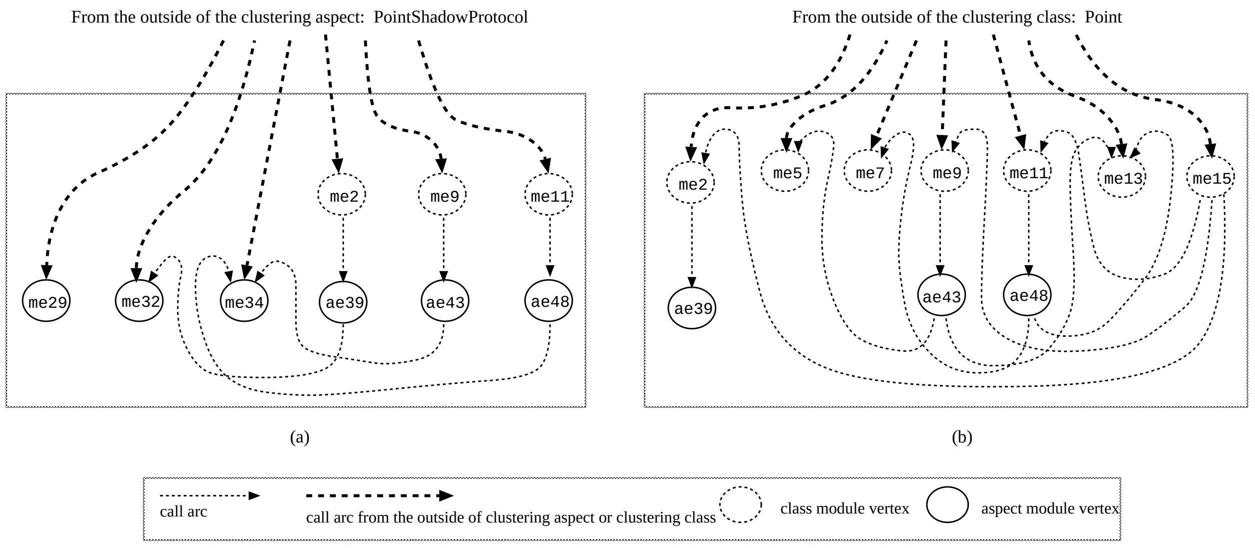
From the outside of the clustering aspect: PointShadowProtocol
me2
me9
me11
me29
me32
me34
ae39
ae43
ae48
(a)
From the outside of the clustering class: Point
me2
me5
me7
me9
me11
me13
me15
ae39
ae43
ae48
(b)
call arc
call arc from the outside of clustering aspect or clustering class
class module vertex
aspect module vertex

Figure 4: The call graphs of c-aspect PointShadowProtocol (a) and c-class Point (b).

entry vertex, *frame loop*, *frame call*, *frame return*, and *frame exit*. The frame call vertex is connected to the *entry vertex* of each CFG for modules. If there is a call in one module to call another module in a c-aspect or c-class, we connect two modules' CFGs at call sites using *call arcs*. The FCFG can simulate a random calling sequence between modules in a c-aspect or c-class, and therefore support the data flow analysis on the c-aspect or c-class. Figures 2 and 3 show the algorithms for constructing the framed control flow graphs for a c-aspect and c-class respectively. According to the algorithms, the FCFG can be constructed by the following three steps.

First, the *call graph* of an c-aspect or c-class is constructed to represent the call relationships among modules in the c-aspect or c-class. It is a digraph such that its vertices represent modules and its arcs represent the calling relations between modules in the c-aspect or c-class. We can modify the existing call graph construction algorithms proposed in [8, 15] to construct the call graph of a c-aspect or c-class. Figure 4 shows the call graphs for c-aspect `PointShadowProtocol` and c-class `Point`.

Second, a frame to represent a test driver for the c-aspect or c-class is constructed and enclosed into the call graph of the c-aspect or c-class to form a partial FCFG. The frame allows one to simulate a random calling sequence to some modules in the c-aspect or c-class.

Third, each vertex v of the call graph in the partial FCFG is replaced with the CFG for v.

For example, Figure 5 shows the FCFGs of c-aspect `PointShadowProtocol` and c-class `Point` respectively.

5 Data Flow Testing of Aspects and Classes

In order to perform testing on c-aspects or c-classes, we present a data-flow-based testing technique to identify modules of a c-aspect or c-class that should be tested. Our data flow testing considers all aspect or class instance variables and def-use pairs that are closed related to some specific program points in the c-aspect or c-class. In this section, we first present a three-level data-flow-based unit testing approach and then describe how to derive the data flow information from c-aspects or c-classes for supporting the testing approach based on the structure model described in Section 4.

5.1 A Three-Level Unit Testing Approach

In aspectJ, aspects and classes are composed of modules such as advice, introduction, and methods, therefore data flow test cases can be derived from three different perspectives, i.e., *intra-module*, *inter-module*, and *intra-aspect* or *intra-class*. For the intra-module perspective, test paths are selected for the variables that have def-use pairs within a module and test cases can be derived from the CFG of the module. For the inter-module perspective, test paths are selected for the variables that have def-use pairs across modules and test cases can be obtained from the ICFG of a group of modules. For the intra-aspect or intra-class perspective, test paths are selected for the variables that have def-use pairs within a c-aspect or c-class and test cases can be derived from the FCFG of a c-aspect or c-class.

From the intra-module, inter-module, and intra-aspect or intra-class perspectives, data flow testing of c-aspects and c-classes can be accomplished through a three-level testing approach. In the following we explain the approach in more

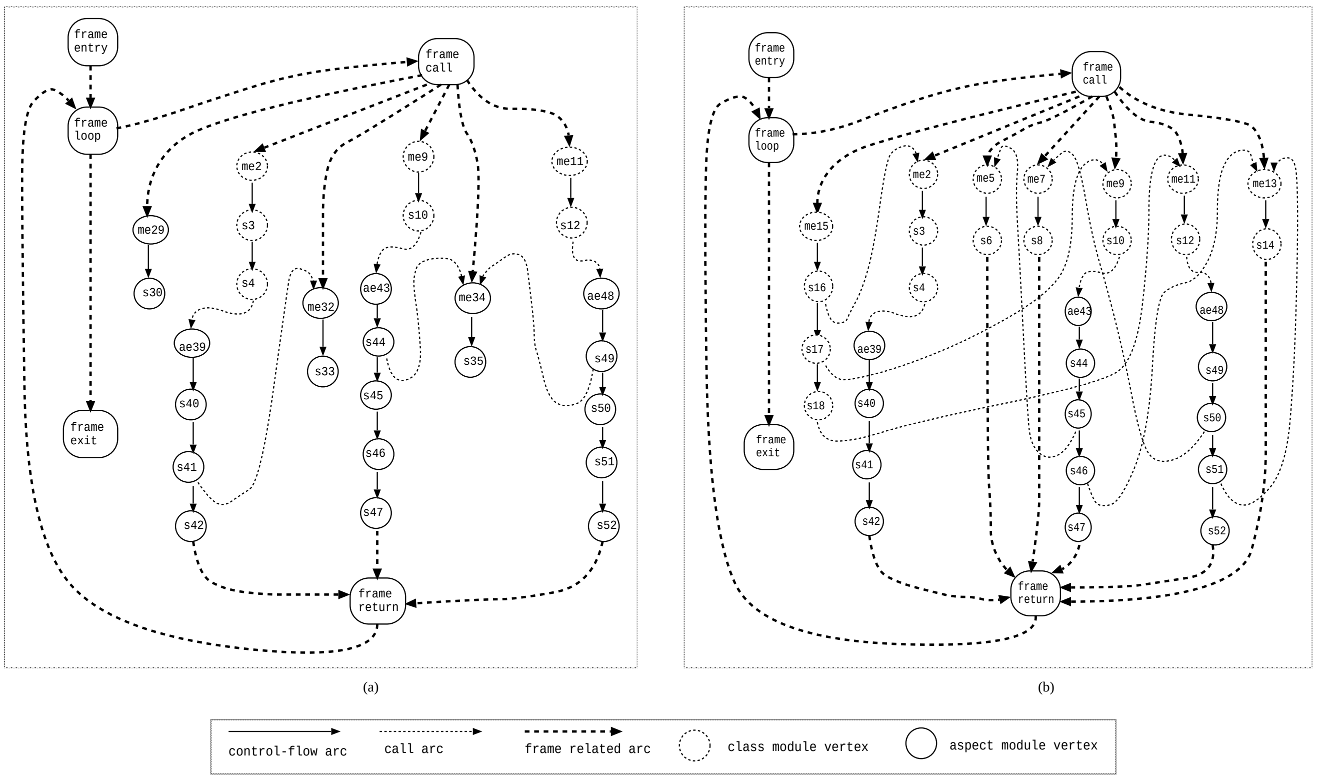

Figure 5: The framed control flow graphs of c-aspect PointShadowProtocol (a) and c-class Point (b).

detail.

Intra-Module Level Testing. *Intra-module testing* is to perform testing on individual modules for the variables that have def-use pairs limited to a single module. For a c-aspect or c-class, intra-module testing has three possible forms: *intra-n-method*, *intra-c-method*, or *intra-introduction* testing. On the other hand, intra-module testing of a c-class has only two forms: *intra-n-method* or *intra-c-method* testing.

Example. For c-aspect `PointShadowProtocol`, intra-module testing includes separately testing of one c-constructor `Point` which contains a piece of after-advice with the `setting` pointcut, and two c-methods `setX` and `setY`, which contains two pieces of after-advice with the `settingX` and `settingY` pointcuts respectively. It also includes separately testing of three n-methods `getShadowCount`, `getShadow`, and `associate` in `PointShadowProtocol`. Intra-module testing of c-class `Point` includes testing of one c-constructor `Point` and two c-methods `setX` and `setY`, that contains three pieces of after-advice with the `setting`, `settingX`, and `settingY` pointcuts respectively, and four n-methods `getX`, `getY`, `printPosition`, and `main` (a special method) in the `Point` class.

Inter-Module Level Testing. *Inter-module testing* is to perform testing on a public module along with some other modules it calls, directly or indirectly, in a c-aspect or c-class for the variables whose def-use pairs involve more than one module within the c-aspect or c-class. Inter-module level testing does not consider invocations from other modules outside the c-aspect or c-class, and aims at testing the internal data interactions among modules within the c-aspect or c-class. For a c-aspect, the interactions among its modules form the interaction chains which are composed of some basic interactions between *c-method* and *c-method*, *c-method* and *introduction*, *c-method* and *n-method*, *introduction* and *introduction*, *introduction* and *n-method*, or *n-method* and *n-method*. For a c-class, the interactions among its modules form the interaction chains which are composed of some basic interactions between *c-method* and *c-method*, *c-method* and *n-method*, or *n-method* and *n-method*.

Example. For c-aspect `PointShadowProtocol`, we can perform inter-module testing on the `setX` c-method (containing the after-advice with the `settingX` pointcut) by integrating the c-method and the `getShadow` n-method in the c-aspect, and testing various calls to the c-method within the aspect. Similarly, we can perform inter-module testing on the `setY` c-method (containing the after-advice with the `settingY` pointcut) by integrating the c-method and the `getShadow` n-method, and testing various calls to the c-method within the aspect.

Intra-Aspect or Intra-Class Level Testing. *Intra-aspect or intra-class testing* is to perform testing on a c-aspect or c-class that has variables whose def-use pairs can be changed by different module invocations from the outside of the c-aspect or c-class. Unlike inter-module level testing that only considers one public module along with other modules it calls within a c-aspect or c-class, intra-aspect or intra-class testing considers multiple modules and their interactions in a c-aspect or c-class, allowing multiple calls to these modules from the outside of the c-aspect or c-class.

Example. For performing intra-aspect testing of c-aspect `PointShadowProtocol`, we may select test sequences such as <c-method `setX`, c-method `setY`, n-method `associate`> and <c-method `setX`, c-method `setY`, n-method `getShadow`>. For performing intra-class testing on c-class `Point`, we may select test sequences such as < c-method `setX`, c-method `setY`, n-method `getX`> and < c-method `setX`, c-method `setY`, n-method `getY`>.

5.2 Computing Def-Use Pairs for Aspects and Classes

In order to perform data flow testing on a c-aspect or c-class of an aspect-oriented program, we need to compute all types of def-use pairs for the c-aspect or c-class, i.e., we need intra-module, inter-module, and intra-aspect or intra-class def-use pairs for the c-aspect or c-class. In the following we borrow the idea from [8] to define three types of def-use pairs for a c-aspect or c-class and show how these types of def-use pairs can be computed based on the structure model described in Section 4.

Intra-Module Def-Use Pairs. Informally, intra-module def-use pairs occur within a single module such as c-method, introduction, and method of a c-aspect or c-class. Intra-module def-use pairs can be used to test the def-use interactions within a single module.

Let A be a c-aspect or c-class, and m be a module of A. Let d and u be two statements in m such that d defines a variable, and u uses the variable. If there exists a program P that calls m such that in P, (d, u) is a def-use pair exercised during a single invocation of m, then (d, u) is a *intra-module def-use pair*.

For example, in c-aspect `PointShadowProtocol`, c-method `setX`, which contains the after-advice with the `settingX` pointcut, has intra-method def-use pair (s44, s45) with respect to variable `s`, because the definition of `s` in statement s44 reaches the use of `s` in statement s45.

Inter-Module Def-Use Pairs. Informally, inter-module def-use pairs occur when modules within the calling context of a single public module interact, such that a definition of a variable in one module reaches across module boundaries to a use of the variable in some module called, directly or indirectly, by the public module in a c-aspect or c-class. Inter-module def-use pairs can be used to test def-use interactions among a public module and a group of modules it calls, directly or indirectly, in the c-aspect or c-class.

Let A be a c-aspect or c-class, and m_0 be a public module of A. Let $\{m_1, m_2, \ldots, m_n\}$ be the set of modules in A called, directly or indirectly, when m_0 is invoked. Let d and u be two statements in m_i and m_j respectively, such that d defines a variable, and u uses the variable, and $m_i, m_j \in \{m_0, m_1, m_2, \ldots, m_n\}$. If there exists a program P that calls m_0 such that in P, (d, u) is a def-use pair exercised during a single invocation by P of m_0, and such that either $m_i \neq m_j$, or m_i and m_j are separate invocation of the same module, then (d, u) is an *inter-module def-use pair*.

For example, in c-aspect `PointShadowProtocol`, c-constructor `Point`, which contains the after-advice with the `setting` pointcut, invokes the `associate` method, and receives a shadow value back, which it uses to initialize the object of class `Shadow`. Def-use pair (s40, s33) is an inter-module pair (between c-method `Point` and method `associate`), because the definition of `s` in statement s40 of the `Point` c-constructor reaches the use of `s` in statement s33 of the `associate` method.

Intra-Aspect and Intra-Class Def-Use Pairs. Informally, intra-aspect or intra-class def-use pairs occur when sequences of public modules are invoked in a c-aspect or c-class.

Let A be a c-aspect or c-class, and m_0 be a public module of A. Let $\{m_1, m_2, \ldots, m_n\}$ be the set of modules in A called, directly or indirectly, when m_0 is invoked. Let n_0 be a public module in A (possibly the same module as m_0), and $\{n_1, n_2, \ldots, n_n\}$ be the set of modules in A called, directly or indirectly, when n_0 is invoked. Let d be a statement in $m_i \in \{m_0, m_1, m_2, \ldots, m_n\}$ and u be a statement in $n_i \in \{n_1, n_2, \ldots, n_n\}$, such that d defines a variable, and u uses the variable. If there exists a program P that calls m_0 and n_0 such that in P, (d, u) is a def-use pair, and such that after d is executed and before u is executed, the call to m_0 terminates, then (d, u) is an *intra-aspect or intra-class def-use pair*.

For example, consider the method sequence <`setX`, `getX`> of c-class `Point`. `setX` may set the value of variable `x` to the value of its formal parameter `_x` passed from a call from the `main()` class, and `getX` may get the value of variable `x` and return it to a call from the after-advice with the `settingX` pointcut. The definition of `x` in statement `s10` of `setX` and the use of `x` in statement `s6` of `getX` form a intra-class def-use pair.

5.2.1 Computing Def-Use Pairs

We can use existing data flow analysis algorithms [7, 17] to compute the intra-module, inter-module, and intra-aspect or intra-class def-use pairs for a c-aspect or c-class of an aspect-oriented program respectively based on three types control flow graphs presented in Section 4

For an aspect-oriented program without the possibility of aliasing [2], we use intraprocedural data flow analysis techniques such as traditional iterative or interval-based data flow analysis [1, 14] to compute the def-use pairs for a single module (i.e., a c-method, a piece of introduction, or a n-method) of a c-aspect or c-class based on the CFG of the module. These techniques operate on the control flow graph of these modules. In order to compute the inter-module def-use pairs for a group of interactive modules in a c-aspect or c-class, and the intra-aspect or intra-class def-use pairs for a c-aspect or c-aspect, we use interprocedural data flow analysis techniques [8] which operate on the ICFG and FCFG.

When an aspect-oriented program contains aliases, we borrow the idea from [7] to use the data flow analysis algorithm developed by Pande, Landi, and Ryder [17] to compute the intra-module, inter-module, and intra-aspect or intra-class def-use pairs for a c-aspect or c-class, based on the FCFG. However, in order to apply the algorithm to the FCFG, some adjustments on the FCFG should be made. We perform the following data flow analysis for a c-aspect or c-class A. First, conditional reaching definitions and conditional alias information for A is computed in terms of the FCFG. Second, the data flow information is propagated through the program using the FCFG and the propagation rules introduced in [17], with the following adjustments by handling (1) the frame call vertex as a call vertex, (2) the frame return vertex as a return vertex, (3) the frame loop vertex as a statement vertex without definitions or uses, and (4) the frame entry and exit vertices as program entry and exit vertices. Through these adjustments and analysis, we can obtain the intra-module, inter-module, and intra-aspect or intra-class def-use pairs for A.

6 Tool Support

To perform data-flow-based unit testing on aspects and classes, we need both control flow and data flow information for each aspect or class being tested. We plan to implement a unit testing tool for AspectJ to realize the approach presented in this paper [21]. Our tool has three components, the *driver generator*, the *compiler*, and the *test case generator*.

The driver generator takes as input some related c-aspects or c-classes and outputs a test driver. This test driver, when executed, reads test cases, check their syntax, executes them, and checks the results. As the first step, currently we generate our test driver by hand.

The compiler takes as input an AspectJ program, and analyze the program to get control flow information such as the predecessors and successors of each statement and caller-callee information, and data flow information such as definition and use of each variable in each statement, which are necessary for constructing control flow graphs and computing def-use pairs for each module, group of modules, and c-aspect or c-class. Based on such information the compiler first constructs the control flow

graph for each module in a c-aspect or c-class, and then constructs the call graph for each c-aspect or c-class. Finally, it constructs the framed control flow graph for each c-aspect or c-class in the program. It also computes the def-use pairs for each module, group of modules, and a single c-aspect or c-class in the program. We plan to modify the AspectJ compiler (`ajc`) for gathering the control flow and data flow information of an AspectJ program.

The test case generator takes as input def-use pairs information outputted by the compiler component, and generates various test cases for each module, group of modules, and a single c-aspect or c-class being tested, which are used by the test driver.

7 Related Work

Although many approaches have been proposed for testing object-oriented programs [4, 5, 8, 11, 18, 20], research on testing aspect-oriented programs has received little attention. To the best of our knowledge, our work presented in this paper is the first to address the problem of testing aspects and classes of aspect-oriented programs.

Harrold and Rothermel [8] propose a method for performing class testing by testing the data flow interactions in a class. In their testing approach, three levels of data flow testing for classes have been proposed, i.e., *intra-method testing*, *inter-method testing*, and *intra-class testing*. Intra-method testing has the same meaning as the unit testing of a procedure in procedural programs. Inter-method testing has the same meaning as the integrating testing of procedures in procedural programs. Intra-class testing performs testing on the interactions of public methods when they are called in random sequences. Their testing method is a program-based one that may provide opportunities to detect errors in classes which may not be uncovered by specification-based class testing.

Buy *et al.* [4] propose a technique for automatic generation of test cases for class testing. They show that how the results of data flow analysis defined by Harrold and Rothermel [8] can be used as part of a method for generating test cases for classes. Their technique is a combination of data flow analysis, symbolic execution, and automatic deduction.

Parrish *et al.* [18] propose an approach to apply the conventional flow graph-based testing strategies to object-oriented class modules. Based on the conventional flow graph, they present a general class graph to represent classes. Based on this new graph, many existing flow graph-based techniques can be applied to classes in both specification-based testing and program-based unit testing.

[2] An *alias* occurs when two names for the same memory location are visible at a point in the program [8].

Although these testing approaches can be used to testing classes from various different viewpoints, they can not be applied directly to test aspects and classes in aspect-oriented programs due to the problems we pointed out in Section 3. Our testing approach can handle the unit testing problems that are unique to aspect-oriented programs.

8 Concluding Remarks

In this paper, we proposed a data-flow-based unit testing approach for aspect-oriented programs. Our unit testing approach tests two types of units for an aspect-oriented program, i.e., *aspects* that are modular units of crosscutting implementation of the program, and those *classes* whose behavior may be affected by one or more aspects. For each aspect or class, our approach performs three levels of testing, i.e., *intra-module*, *inter-module*, and *intra-aspect* or *intra-class* testing. For an individual module such as a piece of advice, a piece of introduction, or a method, or a public module along with other modules it calls in an aspect or class, we perform intra-module or inter-module testing. For modules that can be accessed outside the aspect or class and can be invoked in any order by users of the aspect or class, we perform intra-aspect or intra-class testing. While our three-level testing borrowed some ideas from that of object-oriented programs [8], our approach can handle testing problems that are unique to aspect-oriented programs. We used the control flow graphs as a basis for computing def-use pairs of aspects and classes being tested in an aspect-oriented program and use such information to guide the selection of tests for the aspects or classes.

Our testing approach proposed in this paper focused only on the aspects or classes themselves. We can also, however, apply the data flow testing to the problem of integration testing of aspects and classes. Also, in this paper, we did not consider how to test extended aspect or class (i.e., aspect or class inheritance) in an aspect-oriented program. We would like to study these issues in our future work. We also plan to develop a testing tool based on the technique proposed in this paper to support data flow testing of aspects and classes in AspectJ programs.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. Compiler, Principles, Techniques, and Tools. Addison-Wesley, Boston, MA, 1986.

[2] The AspectJ Team. The AspectJ Programming Guide. August 2001.

[3] L. Bergmans and M. Aksits. Composing crosscutting Concerns Using Composition Filters. *Communications of the ACM*, Vol.44, No.10, pp.51-57, October 2001.

[4] U. Buy, A. Orso, and M. Pezze. Automatic Testing of Classes. *Proc. the International Symposium on Software Testing and Analysis*, pp.39-48, 2000.

[5] R. Doong and P. Frankl. The ASTOOT Approach to Testing Object-Oriented Programs. *ACM Transactions on Software Engineering and Methodology*, Vol.3, No.2, pp.101-130, April 1994.

[6] N. Gupta and R. Gupta. Data Flow Testing. *The Compiler Design Handbook: Optimizations and Machine Code Generation*, CRC Press, 2002.

[7] M. J. Harrold and M. L. Soffa. Efficient Computation of Interprocedural Definition-Use Chains. *ACM Transactions on Programming Languages and Systems*, Vol.16, No.2, pp.175-204, March 1994.

[8] M. J. Harrold and G. Rothermel. Performing Data Flow Testing on Classes. *Proc. ACM SIGSOFT Foundation of Software Engineering*, pp.154-163, December 1994.

[9] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin. Aspect-Oriented Programming. *proc. 11th European Conference on Object-Oriented Programming*, pp220-242, LNCS, Vol.1241, Springer-Verlag, June 1997.

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin, "An Overview of AspectJ," *proc. 13th European Conference on Object-Oriented Programming*, pp.220-242, LNCS, Vol.1241, Springer-Verlag, June 2000.

[11] D. Kung, J. Gao, P. Hsia, and Y. Toyoshima, C. Chen, K.-S. Kim, and Y.-K. song. Developing an Object-Oriented Software Testing and Maintenance Environment. *Communications of ACM*, Vo.38, No.10, pp.75-86, October 1995.

[12] J. Laski and B. Korel. A Data Flow Oriented Program Testing Strategy. *IEEE Transaction on Software Engineering*, Vol.9, pp.33-43, May 1983.

[13] K. Lieberher, D. Orleans, and J. Ovlinger. Aspect-Oriented Programming with Adaptive Methods. *Communications of the ACM*, Vol.44, No.10, pp.39-41, October 2001.

[14] S. S. Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmann, 1997.

[15] D. Sereni and O. de Moor. Static Analysis of Aspects. *Proc. 2nd International Conference on Aspect-Oriented Software Development*, pp.30-39, March 2003.

[16] P. Tarr, H. Ossher, W. H. Harrison, and S. M. Sutton. N Degrees of Separation of Concerns: Multi-Dimensional Separation of Concerns. *Proc. 21th International Conference on Software Engineering*, pp.107-119, May 1999.

[17] H. Pande, W. Landi, and B. G. Ryder. Interprocedural Def-Use Associations in C Programs. *IEEE Transaction on Software Engineering*, Vol.20, No.5, pp.385-403, May 1994.

[18] A. S. Parrish, R. B. Borie, and D. W. Cordes. Automated Flow Graph-Based Testing of Object-Oriented Software Modules. *Journal of Systems and Software*, Vol.20, pp.95-109, 1993.

[19] S. Rapps and E. J. Weyuker. Selecting Software Test Data Using Data Flow Information. *IEEE Transaction on Software Engineering*, Vol.11, No.4, pp.367-375, April 1985.

[20] C. D. Turner and D. J. Robson. The State-Based Testing of Object-Oriented Programs. *Proc. International Conference on Software Maintenance*, pp.302-310, September 1993.

[21] J. Zhao. Tool Support for Unit Testing of Aspect-Oriented Software. *OOPSLA'2002 Workshop on Tools for Aspect-Oriented Software Development*, Seattle, WA, USA, November 2002.

[22] H. Zhu, P. A. V. Hall, and J. H. R. May. Software Unit Test Coverage and Adequacy. *ACM Computing Surveys*, Vol.29, No.4, pp.366-427, December 1997.