2003－SE－142 (5)
2003／5／30

Unit Testing for 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

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.,* module, inter-module, *and* aspect or class *testing. For an individual module such as advice, introduction, and a method, or a public module along with other modules it calls in an aspect or class, we perform 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 aspect or class testing.*

1 Introduction

Aspect-oriented software development (AOSD) is a new technique to support separation of concerns in software development [3, 8, 11, 13]. 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 programs differs significantly from procedural and object-oriented programs in terms of analysis, design, structure, and development techniques. In aspect-oriented programs, the basic unit of organization is the aspect (or class) construct. An aspect with its encapsulation of state with associated advice, introduction, and methods (operations) is a significantly different abstraction in comparison to the procedure or class unit within procedural or object-oriented programs. The inclusion of join points in an aspect where pieces of code can be advised or introduced to one or more classes further complicates the static and dynamic relationships among aspects and classes. These specific features in aspect-oriented programs require special testing support and also provide opportunities for exploitation by a testing strategy. However, although many testing approaches have been proposed for procedural and object-oriented programs, they 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 [19]. 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)*. 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* [7, 16] 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.

This paper proposes 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.

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) along. 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.

Based on the above consideration, our unit testing approach tests two types of units in 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., *module*, *inter-module*, and *aspect or 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 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 aspect or class testing. Our approach can handle unit testing problems that are unique to aspect-oriented programs. We use the control-flow graph as a basis for computing def-use pairs of an aspect or class being tested and use such information to perform data-flow testing on the aspect or class.

The rest of the paper is organized as follows. Section 2 briefly introduces the AspectJ and data-flow testing of object-oriented programs. Section 3 discusses some issues that arise in testing aspects or classes in aspect-oriented programs. Section 4 describes a data-flow based approach to testing of aspects or classes. testing approach. Section 5 discusses related work. Concluding remarks are given in Section 6.

2 Background

2.1 AspectJ

In this paper we use AspectJ [2] as our target language to show the basic ideas of our unit testing approach for aspect-oriented programs. We believe that our ideas are independent of AspectJ and are generally applicable to the class of aspect-oriented programming languages.

Aspect-oriented programming (AOP) is a programming technique for expressing programs involving encapsulated, crosscutting concerns through composition techniques, and through reuse of the crosscutting code [2, 13]. AspectJ is a seamless aspect-oriented extension to Java by adding some new concepts and associated constructs to Java. These concepts and associated constructs are called join points, pointcut, advice, introduction, and aspect.

Aspect is modular unit of crosscutting implementation in AspectJ. Each aspect encapsulates functionality that crosscuts other classes in a program. An aspect is defined by aspect declaration, which has a similar form of class declaration in Java. Similar to a class, an aspect can be instantiated and can contain state and methods, and also may be specialized in its sub-aspects. An aspect is then combined with the classes it crosscuts according to specifications given within the aspect. Moreover, an aspect can *introduce* methods, attributes, and interface implementation declarations into types by using an *introduction* construct. 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 members.

In addition to introductions, the essential mechanism provided for composing 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, 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 a complete listing of possible designators one can refer [2].

An aspect can specify *advice* that is used to define some code that is executed when a pointcut is reached. Advice is a method-like mechanism which consists of instructions that is executed *before*, *after*, or *around* a pointcut. `around` advice executes *in place* of the indicated pointcut, allowing a method to be replaced.

Example. Figure 1 shows a sample AsepctJ program taken from [2] that associates shadow points with every `Point` object and contains one `PointShadowProtocol` aspect that stores a shadow object in every `Point` and two classes `Point` and `Shadow`.

2.2 Data Flow Testing of Object-Oriented Programs

Data-flow testing is to test how values which are associated with variables can affect the execution of the program. Data-flow testing is concerned with the variable occurrences within the program. Each variable occurrence is classified as either a definition occurrence or an use occurrence. A definition occurrence of a variable is where a value of the variable is defined. A use occurrence of a variable is where the value of the variable is used. Use occurrence can be further 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 occurrence of the variable 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 occurrence is called *predicate use*, denoted as *p-use* [19].

Data-flow testing has been applied to test classes in object-oriented programs [7]. In such a testing, 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.

In order to perform data-flow testing on classes at three different levels, one must compute three kinds of def-use

```
  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.

pairs, i.e., intra-method, inter-method, and intra-class def-use pairs in a class, that correspond to these three levels of testing. Intra-method def-use pairs have the same meaning as that in procedural programs, that is, a definition of a variable and a subsequent use are both located in the same method. Techniques for performing intraprocedural data-flow analysis can be used for intra-method data-flow analysis of primitive types [14]. Inter-method def-use pairs have the same meaning as inter-procedural def-use pairs as that have in procedural programs. Intra-class def-use pairs arise due to sequences of method invocations that arise if the class was instantiated. An instantiated class can call methods in any order. In order to compute intra-class def-use pairs, three program representations, i.e., a *class call graph*, a *frame* around the class call graph, and a *class control-flow graph* are needed. The flame allows users to simulate a random calling sequence between methods in a class, and enables techniques for interprocedural def-use analysis to be applied to detect def-use pairs of primitive types in the class with different levels of precision due to aliasing effects and the techniques used for dealing with aliases [7].

3 Unit Testing of Aspect-Oriented Programs

We next present a motivation example to discuss the issues that arise in testing aspects and classes of aspect-oriented programs, and discusses several kinds of unit testing problems.

3.1 Motivation Example

Consider the aspect `PointShadowProtocol` shown in Figure 1, that modifies the behavior of class `Point`. `PointShadowProtocol` declares a piece of *after-advice* (with the `settingX` pointcut), or code to be executed before traversing a join point into a method body. This after-advice is applicable 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 specify 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 unit testing on `PointShadowProtocol`, we must 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 must (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 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.

In this paper, we focus on testing c-aspect and c-class units in an aspect-oriented program, and do not consider n-classes of the program because these n-classes can be tested using existing class testing techniques [7] for object-oriented programs.

Example. Consider the program shown in Figure 1, which is composed of one aspect `PointShadowProtocol` and two classes `Point` and `Shadow`. 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 the c-aspect, c-class, or n-class of its original aspect or class, and also the same name to represent the c-method or n-method of its original method.

[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*.

In the rest of the paper, for unification, we use the word "module" to stand for a c-method, a piece of introduction, or a n-method declared in an aspect or class.

3.2 Types of Unit Testing for Aspects and Classes

When performing aspect or class testing on aspect-oriented programs, we consider three different kinds of unit testing problems for a c-aspect or a-class, i.e., *module*, *inter-module*, and *aspect or class* testing. Performing these kinds of testing on a c-aspect or c-class generally involves a module or a group of modules in the aspect or class. In all cases of the testing, we may need some testing stubs and/or drivers to perform the testing, and also techniques to inspect the state of the aspect or class (object) after the module sequence invocation. Below, we explain each of these testing problems with examples.

Module Testing. *Module testing* is to perform testing on an individual module in a c-aspect or c-class. For a c-aspect or c-class, module testing has three possible forms: *n-method*, *c-method*, and *introduction* testing, and has the same meaning as intra-method testing of object-oriented programs [7], because n-method, c-method, or introduction can be regarded as a method-like module. On the other, module testing of a c-class has only two forms, i.e, *n-method* and *c-method* testing.

Example. Module testing of c-aspect `PointShadowProtocol` includes separately 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 three n-methods `getShadowCount`, `getShadow`, and `associate` in `PointShadowProtocol`. Module testing of 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` (special method) in the `Point` class.

Inter-Module 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. Inter-module testing does not consider invocations from other modules outside the c-aspect or c-class, and aims at testing the internal 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*, and *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*, and *n-method* and *n-method*.

Example. Consider 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.

Aspect or Class Testing. *Aspect or Class testing* is to test the interactions of multiple public modules in a c-aspect or c-class when they are called in a random sequence from the outside of the c-aspect or c-class. Unlike inter-module testing that only considers one public module along with other modules it calls within a c-aspect or c-class, aspect or 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. Consider aspect testing on the `PointShadowProtocol` c-aspect, 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 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`>.

4 Data Flow Testing of Aspects and Classes

In order to perform testing on aspects or classes, we present data-flow based testing technique to identify modules of an 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 define three types of def-use pairs and then describe how to compute these def-use pairs for the c-aspect or c-class based on a framed control-flow graph.

4.1 Def-Use Pairs for Aspects and Classes

We identify the *module*, *inter-module*, and *aspect or class* def-use pairs in a c-aspect or c-class that should be tested to correspond to the three levels of c-aspect or c-class testing described in Section 3. In the following we generalize the definitions of three def-use pairs for a class in an object-oriented program described in [7] to define the three types of def-use pairs for a c-aspect or c-class in an aspect-oriented program and describe them with examples.

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

Definition 1 *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 an module def-use pair.*

Example. In c-aspect `PointShadowProtocol`, c-method `setX`, which contains the after-advice with the `settingX` pointcut, has 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.

Definition 2 *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.*

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.

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

Definition 3 *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 aspect or class def-use pair.*

Example. Consider the method sequence <`setX`,`getX`>. `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 class def-use pair.

4.2 Computing Def-Use Pairs for Aspects and Classes

In order to perform data-flow testing on a c-aspect or c-class, we need to compute all types of def-use pairs for the c-aspect or c-class, i.e., we need module, inter-module, and aspect or class def-use pairs for the c-aspect or c-class. In the following, we first describe how to construct the framed control-flow graph and then show how to compute the three types of def-use pairs based on the graph.

Framed Control Flow Graphs for Aspects and Classes. We use the *framed control-flow graph* (FCFG) to compute the def-use pairs of a c-aspect or c-class. The FCFG consists of a collection of *control-flow graphs* (CFGs) each presents a module in a 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 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. 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.

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

Example. Figure 2 shows the call graphs for c-aspect `PointShadowProtocol` and c-class `Point`. Figure 3 shows the FCFGs of c-aspect `PointShadowProtocol` and c-class `Point` respectively.

Computing Def-Use Paris. Having the FCFG as a representation for a c-aspect or c-class of an aspect-oriented program, we can use existing data-flow analysis algorithms [6, 14] to compute the module, inter-module, and aspect or class def-use pairs for each c-aspect or c-class of the program based on the FCFG.

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, 12] to compute the def-use pairs for a single c-method, introduction, or n-method in a c-aspect or c-class. 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 aspect or class def-use pairs for a c-aspect or c-aspect, we use interprocedural data flow analysis techniques [7].

When an aspect-oriented program contains aliases, we borrow the idea from [6] to use the data flow analysis algorithm developed by Pande, Landi, and Ryder [14] to compute the module, inter-module, and aspect or 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 [14], 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 module, inter-module, and aspect or class def-use pairs for A.

5 Related Work

We discuss some related work on unit testing of object-oriented programs which directly or indirectly influence our work on aspect-oriented programs. We focus on comparing our work with the program-based unit testing of object-oriented programs. For specification-based class testing, one can refer to [5, 10, 17].

Harrold and Rothermel [7] propose a method for performing class testing by testing the data-flow interactions in a class. The detailed for their testing method has been described in Section 2. 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 [7] 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. Roughly speaking, our unit testing approach for aspect-oriented programs can be regarded as an extension of the testing approach proposed by Harrold and Rothermel [7] to handle the unit testing problems that are unique to aspect-oriented programs. It is also possible to further explore the problem of generating test cases for c-aspect or c-class in aspect-oriented programs based on the technique proposed by Buy *et al.* [4].

Parrish *et al.* [15] propose an approach to apply the

[2]An *alias* occurs when two names for the same memory location are visible at a point in the program [7].

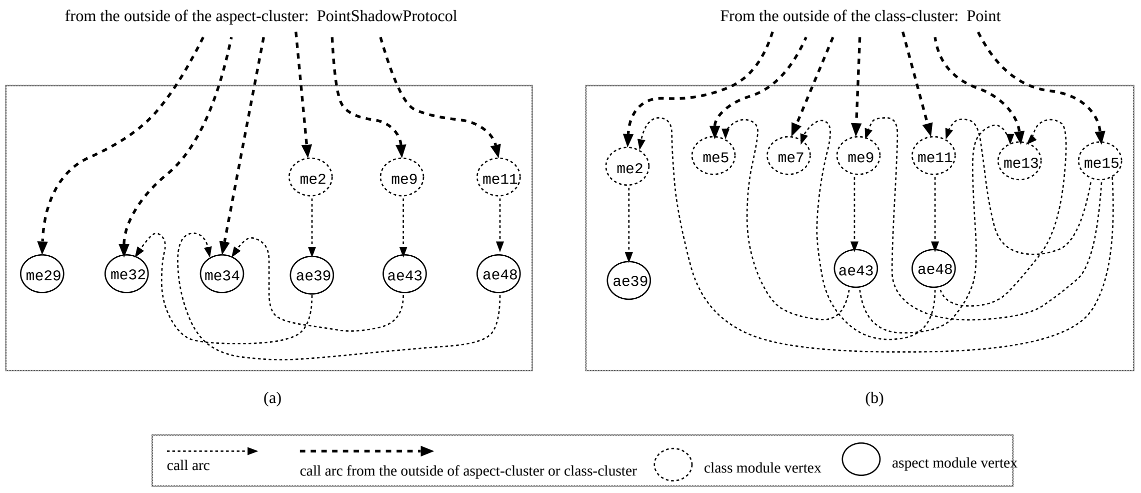
from the outside of the aspect-cluster: PointShadowProtocol
From the outside of the class-cluster: Point
me2
me9
me11
me29
me32
me34
ae39
ae43
ae48
(a)
me2
me5
me7
me9
me11
me13
me15
ae39
ae43
ae48
(b)
call arc
call arc from the outside of aspect-cluster or class-cluster
class module vertex
aspect module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

Figure 2: The call graphs of c-aspect PointShadowProtocol (a) and c-class Point (b).

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. They also show their insights on how to define a new set of test coverage criteria in terms of the class graph. However, in contrast to the approach proposed by Parrish *et al.* [15], that considers the form of data flow information involving types during testing, we consider the data-flow information with variables. Moreover, with some extensions of the class graph, we can use a similar way as Parrish *et al.* [15] used to test aspect-oriented programs.

Kim and Wu [9] focus on the issue related to data binding in class testing. Their class testing has three phases. First, each method in the class is tested by using existing functional and structural testing approaches. Second, actual data bindings are tested with the focus of the data binding between methods. Finally, the sequences of methods are tested by using the class graph model proposed by Parrish *et al.* [15] as a basis. To do so, they divide the class graph into a set of sub-graphs in terms of the data members in the class. Based on these sub-graphs, they use the different flow-graph test generation methods given in [15] to achieve various flow graph-based testing criteria. However, as we pointed out in Section 3, when testing aspects or classes in aspect-oriented programs, it is impractical to test them in isolation, and we should consider not only the class being tested, but also those aspects that may affect the class's behavior.

In summary, although the class testing approaches discussed above can be used to testing classes from various different viewpoints, they can not, however, be applied directly to testing aspects or classes in aspect-oriented programs due to the problems we pointed out in Section 3. To the best of our knowledge, our work presented in this paper is the first time to address the problem of testing aspects and class of an aspect-oriented program.

6 Concluding Remarks

In this paper, we proposed a data-flow based approach to testing 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., *module*, *inter-module*, and *aspect or 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 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 aspect or class testing. While our three-level testing borrowed some ideas from that of object-oriented programs [7], our approach can handle testing problems that are unique to aspect-oriented programs.

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 are planing to develop a unit testing tool [18] 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. `http://aspectj.org`

[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.

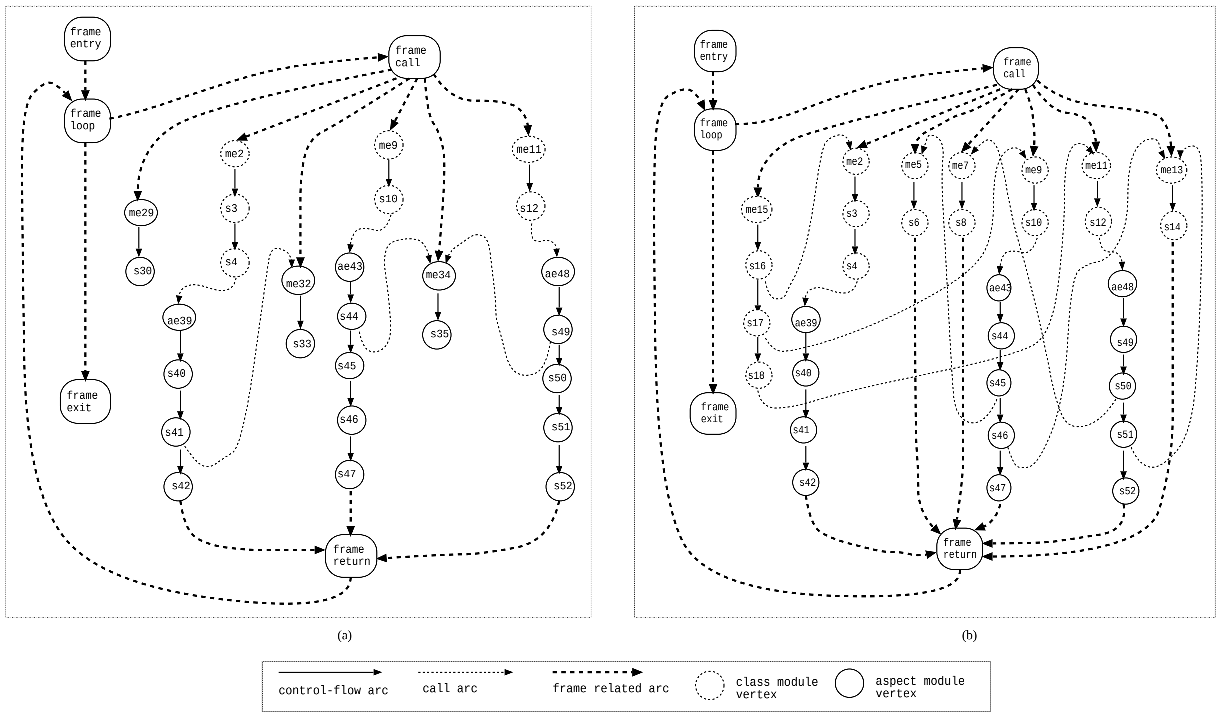

Figure 3: The framed control-flow graphs of c-aspect PointShadowProtocol (a) and c-class Point (b).

[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] 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.

[7] M. J. Harrold and G. Rothermel. Performing Data Flow Testing on Classes. *Proc. ACM SIGSOFT Foundation of Software Engineering*, 1994.

[8] 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.

[9] H. Kim and C. Wu. A Class Testing Technique Based on Data Binding. *Proc. 1996 Aisa-Pacific Software Engineering Conference*, pp.104-109, 1996.

[10] 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.

[11] 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.

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

[13] H. Ossher and P. Tarr. Multi-Dimensional Separation of Concerns and the Hyperspace Approach. *Proc. the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*, Kluwer, 2001.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.