Slicing Aspect-Oriented Software

Jianjun Zhao
Department of Computer Science and Engineering
Fukuoka Institute of Technology
zhao@cs.fit.ac.jp

Abstract

In this paper, we propose an approach to slicing aspect-oriented software. To solve this problem, we present a dependence-based representation called aspect-oriented system dependence graph (ASDG), which extends previous dependence graphs, to represent aspect-oriented software. The ASDG of an aspect-oriented program consists of three parts: (1) a system dependence graph for non-aspect code, (2) a group of aspect dependence graphs for aspect code, and (3) some additional dependence arcs used to connect the system dependence graph to the aspect dependence graphs. After that, we show how to compute a static slice of an aspect-oriented program based on the ASDG.

1 Introduction

1.1 Program Slicing

Program slicing, originally introduced by Weiser [17], is a decomposition technique which extracts program elements related to a particular computation from a program. A *program slice* consists of those parts of a program that may directly or indirectly affect the values computed at some program point of interest, referred to as a *slicing criterion*. The task to compute program slices is called *program slicing*.

Program slicing has been studied primarily in the context of procedural programming languages [15]. In such languages, slicing is typically performed by using a control flow graph or a dependence graph [10, 14]. Program slicing has many applications in software engineering activities including program understanding [7], debugging [1], testing [3], maintenance [9], reuse [13], reverse engineering [4], and complexity measurement [14]. Recently program slicing has been also applied to object-oriented software to handle various object-oriented features [6, 12, 16, 19].

1.2 Aspect-Oriented Programming with AspectJ

Aspect-oriented programming has been proposed as a technique for improving separation of concerns in software design and implementation [11]. Aspect-oriented programming works by providing explicit mechanisms for capturing the structure of crosscutting concerns in software systems.

AspectJ [2] is a seamless aspect-oriented extension to Java. It can be used to cleanly modularize the crosscutting structure of concerns such as exception handling, synchronization, performance optimizations, and resource sharing, that are usually difficult to express cleanly in source code using existing programming techniques. AspectJ can control such code tangling and make the underlying concerns more apparent, making programs easier to develop and maintain.

AspectJ adds some new concepts and associated constructs to Java. These concepts and associated constructs are called join points, pointcuts, advice, and aspect. The *join point* is essential element in the design of any aspect-oriented programming language since join points are the common frame of reference that defines the structure of crosscutting concerns. The join point in AspectJ are well-defined points in the execution of a program. In AspectJ, there are nine types of join points, that is, *method call, constructor call, method execution, constructor execution, object initialization, class initialization, field reference, field assignment*, and *handler execution*. *Advice* is a method-like mechanism used to define certain code that is executed when a pointcut is reached. There are three types of advice, that is, *before, after*, and *around*. In addition, there are also two special cases of after advice, *after returning* and *after throwing*, corresponding to the two ways a sub-computation can return through a join point. *Aspects* are modular units of crosscutting implementation. Aspects are defined by aspect declarations, which have a similar form of class declarations. Aspect declarations may include pointcut declarations, advice declarations, as well as other declarations such as method declarations, that are permitted in class declarations. An AspectJ program is composed of two parts: (1) *non-aspect code* part which includes some classes, interfeces, and other language constructs as in Java, (2) *aspect code* part which includes aspects for modeling crosscutting concerns in the program. Moreover, any implementation of AspectJ is to ensure that aspect and non-aspect code run together in a properly coordination fashion. Such a process is called *aspect weaving* and involves making sure that applicable advice runs at the appropriate join points. For detailed information about AspectJ, one can refer to [2]. In this paper, we will use AspectJ (version 1.0b) as our target language to show the basic idea of our slicing approach.

Example. Figure 1 shows a sample AspectJ program taken from [2]. The program associates shadow points with every `Point` object, and contains a `Point` class, a `Shadow` class, and a `PointShadowProtocol` aspect that stores a shadow object in every `Point`.

1.3 Why Slicing Aspect-Oriented Software ?

Aspect-oriented programming languages present unique opportunities and problems for program analysis schemes such as program slicing. For example, to perform slicing on aspect-oriented software, specific aspect-oriented features such as joint point, pointcut, advice, and aspect, that are different from existing procedural or object-oriented programming languages, must be handled appropriately. Moreover, although these specific features provide the great strengths for aspect-oriented languages to model the crosscutting concerns in an aspect-oriented program, they also introduce difficulties to program analysis tasks.

However, we found that although a number of approaches have been proposed for slicing procedural or object-oriented software, there is no, until recently, slicing algorithm for aspect-oriented software, and due to the specific aspect-oriented features, existing slicing al-

```
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 PointShodowProtocol.
                    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) returning(Point p):
            setting(Point p) {
  s40     Shadow s = new Shadow(x,y);
  s41     associate(p,s);
  s42     shadowCount++;
        }
 ae43   after(Point p): settingX(Point 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(Point p) {
  s49     Sahdow s = getShadow(p);
  s50     s.y = p.getY() + Shadow.offset;
  s51     p.printPosition();
  s52     s.printPosition();
       }
     }
```

Figure 1: A sample AspectJ program.

gorithms for procedural or object-oriented software can not be applied to aspect-oriented software straightforwardly.

1.4 Our Approach to Slicing Aspect-Oriented Software

In this paper, we present a slicing approach for aspect-oriented software. To solve this problem, we develop a dependence-based representation called *aspect-oriented system dependence graph*, that extends previous dependence graphs, to represent aspect-oriented software. The aspect-oriented system dependence graph consists of three parts: (1) a system dependence graph for non-aspect code, (2) a group of aspect dependence graphs for aspect code, and (3) some additional dependence arcs used to connect the system dependence graph to the aspect dependence graphs. After that, we show how to compute a static slice of an aspect-oriented program based on the aspect-oriented system dependence graph. Our main contribution in this paper is a new dependence-based representation, namely, aspect-oriented system dependence graph for aspect-oriented software on which static slices of aspect-oriented software can be computed efficiently.

The rest of the paper is organized as follows. Section 2 presents the aspect-oriented system dependence graph for aspect-oriented software. Section 3 shows how to compute static slices based on the graph. Concluding remarks are given in Section 4.

2 The Aspect-Oriented System Dependence Graph

Aspect-oriented programming languages differ from procedural or object-oriented programming languages in many ways. Some of these differences, for example, are the concepts of joint points, advice, aspects, and their associated constructs. These aspect-oriented features may impact on the development of dependence-based representation for aspect-oriented software, and therefore should be handled appropriately.

In this section we present the *aspect-oriented system dependence graph* (ASDG) to represent aspect-oriented software. An ASDG of an aspect-oriented program consists of three parts: (1) a *system dependence graph* (SDG) for non-aspect code, (2) a group of aspect dependence graphs for aspect code, and (3) some additional dependence arcs used to connect the system dependence graph to the aspect dependence graphs.

The construction of the ASDG of an aspect-oriented program consists of four phases:

(1) Constructing the SDG for non-aspect code of the program, by using existing techniques for object-oriented software.

(2) Constructing the aspect dependence graphs for aspect code of the program.

(3) Determining weaving-points in non-aspect code and inserting weaving-vertices into the SDG.

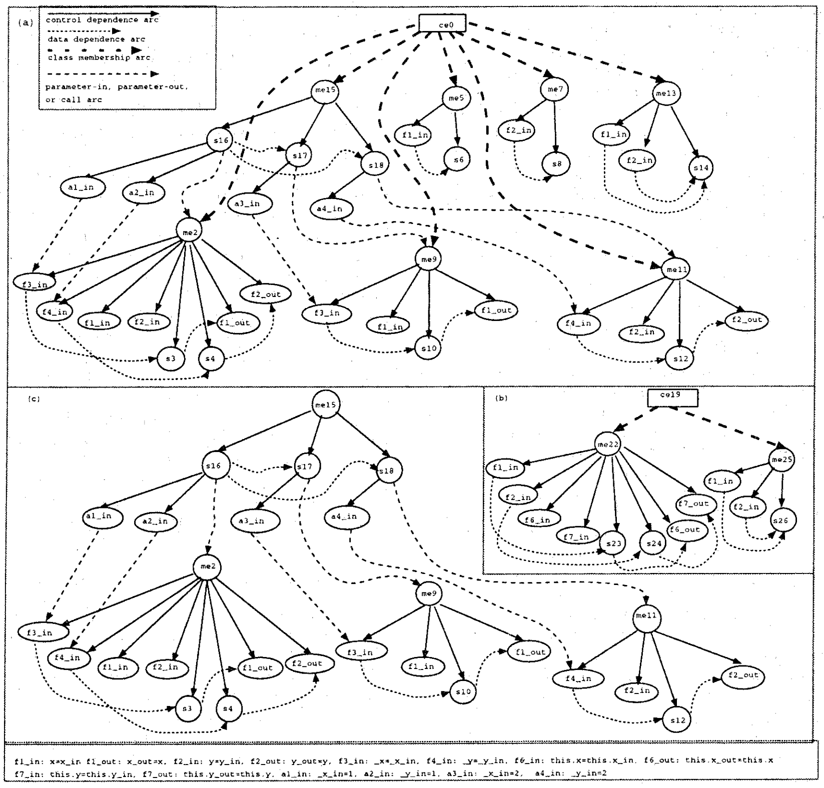
(a)
control dependence arc
data dependence arc
class membership arc
parameter-in, parameter-out, or call arc
(b)
(c)
ce0
ce19
f1_in: x=x_in f1_out: x_out=x, f2_in: y=y_in, f2_out: y_out=y, f3_in: _x=_x_in, f4_in: _y=_y_in, f6_in: this.x=this.x_in, f6_out: this.x_out=this.x
f7_in: this.y=this.y_in, f7_out: this.y_out=this.y, a1_in: _x_in=1, a2_in: _y_in=1, a3_in: _x_in=2, a4_in: _y_in=2

Figure 2: (a) A CDG for class Point, (b) A CDG for class Shadow, and (c) A SDG for non-aspect code of the program in Figure 1.

(4) Weaving the SDG and the ADGs at weaving vertices to form the ASDG by adding some special kinds of dependence arcs between the SDG and each of the advice dependence graphs.

In the rest of this section, we give our construction algorithm in more detail.

2.1 Representing Non-Aspect Code

The first part of our algorithm is to construct the SDG for non-aspect code of an aspect-oriented program. Since the non-aspect code of an AspectJ program is similar to a Java program, in this paper, we use the *Java system dependence graph* (JSDG) [18] originally developed for Java software, to represent the non-aspect code of an AspectJ program.

The JSDG is an extension of the *system dependence graph* (Larsen-Harrold SDG)[12] proposed for representing object-oriented software such as C++, to the case of Java software. However, it differs from the Larsen-Harrold SDG in that in addition to common object-oriented features such as classes and objects, class inheritance, polymorphism, and dynamic binding, it can also represent some specific features in Java such as interface and package. Therefore, we can use the JSDG to represent non-aspect code of an AspectJ program. For the rest of the paper, we use the terms "SDG" and "JSDG" interchangeably.

The SDG for the non-aspect code is a collection of method dependence graphs each representing a `main()` method, or a method in a class of the program, and some additional arcs to represent direct program dependencies between a call and the called method and transitive interprocedural data dependencies.

First, we use the *method dependence graph* proposed in [18] to represent a method of a class. The method dependence graph is an arc-classified digraph whose vertices are connected by several types of dependence arcs. The vertices of the method dependence graph represent statements or control predicates of conditional branch statements in the method. There is an unique vertex called *method start vertex* to represent the entry of the method. In order to model parameter passing between methods, a method dependence graph also includes formal parameter vertices and actual parameter vertices. At the method entry there is a *formal-in vertex* for each formal parameter of the method and a *formal-out vertex* for each formal parameter that may be modified by the method. At each call site there is an *actual-in vertex* for each actual parameter at call site and an *actual-out vertex* for each actual parameter that may be modified by the called method. In addition, at each call site of the method, a *call vertex* is created for connecting the called method. The arcs of the method dependence graph represent two types of dependence relationships, i.e., *control dependence*, and *data dependence*. There is a control dependence arc between two vertices u and v if u is control dependent on v, and there is a data dependence arc between two vertices u and v if u is data dependent on v. In addition, each formal parameter is control dependent on the method start vertex, and each actual parameter is control dependent on the call statement.

Second, we use the *class dependence graph* (CDG) proposed in [12, 18] to represent a single class in the non-aspect code. The CDG of a class is an arc-classified digraph which consists of a collection of method dependence graphs each representing a single method in the class, and some additional vertices and arcs to model parameter passing between different methods in the class. There is an unique *class start vertex* for the class to represent the entry of the class, and the class start vertex is connected to the method start vertex of each method in the class by *class-membership arcs*. If a method invokes another method in the class, the method dependence graphs of two methods are connected at call site. In such a case, a call arc is added between a call vertex of a method and the method start vertex of the method dependence graph of the called method, and *parameter arcs* are added to connect actual-in and formal-in vertices, and formal-out and actual-out vertices to model parameter passing between the methods in the class. Moreover, similar to [10], we use *summary arcs* to represent the *transitive flow of dependencies* in the CDG. Transitive flow of dependence occurs between an actual-in vertex and an actual-out vertex if the value associated with the actual-in vertex affects the value associated with the actual-out vertex. The transitive flow of dependence can be caused by data dependence, control dependence, or both. A *summary* arc models the transitive flow of dependence across a procedure call.

In addition to the things mentioned above, the CDG for a class can also represent the effects of *return* statements. A *return* statement leads to a method to return a value to its caller. In a CDG, a vertex for each *return* statement is connected to its corresponding call vertex by a parameter-out arc. Moreover, if a actual-in parameter may affect the returned value, we add a summary arc between the actual-in vertex and the call vertex. This kind of summary arcs facilitate interprocedural slicing.

Finally, the construction of the SDG for the non-aspect code can be performed by connecting calls in the partial SDG to methods in the CDG for each class. It contains connecting call vertices to the method start vertices to form call arcs, actual-in vertices to formal-in vertices to form parameter-in arcs, and formal-out vertices to actual-out vertices to form parameter-out arcs. Summary arcs for methods are added in a previously analyzed class between the actual-in and actual-out vertices at call sites. Moreover, to create the SDG, the method dependence graph for the `main()` method is connected with other methods in classes of the non-aspect code at call sites. A call arc is added between a method call vertex and the method start vertex of the method dependence graph of the called method. Actual-in and formal-in vertices are connected by parameter-in arcs, and formal-out and actual-out vertices are connected by parameter-out arcs.

Example. Figure 2 (a) shows the CDG for class `Point`, and Figure 2 (b) shows the CDG for class `Shadow` of the program in Figure 1. Figure 2 (c) shows the SDG for non-aspect code of the program in Figure 1.

2.2 Representing Aspect Code

The second part of our algorithm is to construct the ADG for aspect code in an aspect-oriented program. An aspect in AspectJ is a modular unit of crosscutting implementation. Its definition is very similar to a class in Java, and can contain methods, fields, and initializers. Implementation of crosscutting concerns can be done by using pointcuts and advice, and only aspect may include advice, making AspectJ be able to define crosscutting effects. The declaration of those effects is localized. In general, each aspect is composed of an association with other program elements that may include ordinary variable, methods, pointcuts, introductions, and advice, where advice may be before, after, and around. In this subsection, we first describe how to construct the advice dependence graph for advice and the introduction dependence graph for an introduction. We then show how to construct the aspect dependence graph for an aspect that may consists of advice, introductions, pointcuts, and methods. Finally, we show how to represent interactions among aspects and classes.

The Advice Dependence Graph

Advice in AspectJ is a method-like mechanism used to define certain code that is executed when a pointcut is reached. We use advice dependence graph to represent advice in an aspect. The *advice dependence graph* is an arc-classified digraph such that its vertices represent a statement or a control predicate of a conditional branch statement in the advice, and its arcs represent control or data dependence relationships between these statements as in a method dependence graph. There is an unique vertex called *advice start vertex* to represent the entry of the advice.

To model parameter passing, we create a formal-in or formal-out vertex at each advice start vertex (i.e., there is a formal-in vertex for each formal parameter of the advice and there is a formal-out vertex for each formal parameter that may be modified by the advice), and an actual-in or actual-out vertex at each call site in the advice (i.e., there is an actual-in vertex for each actual parameter of the advice, and an actual-out vertex for each actual parameter that may be modified by the advice). We also create a call vertex at each call site of the advice. Moreover, each actual parameter vertex is control dependent on the call vertex of the advice, and each formal parameter vertex is control dependent on the advice start vertex. Finally, we treat instance variables at the advice entry and call sites in the advice as parameters. Therefore, we should also create actual-in, actual-out, formal-in, formal-out vertices for these variables.

The Introduction Dependence Graph

In AspectJ, an introduction can add whole new elements (i.e., fields, methods, or constructors) in the given types (i.e., classes). We use an introduction dependence graphs to represent an introduction in an aspect. The *introduction dependence graph* is an arc-classified digraph such that its vertices represent a statement or a control predicate of a conditional branch statement in the introduction, and its arcs represent control or data dependence relationships between these statements as

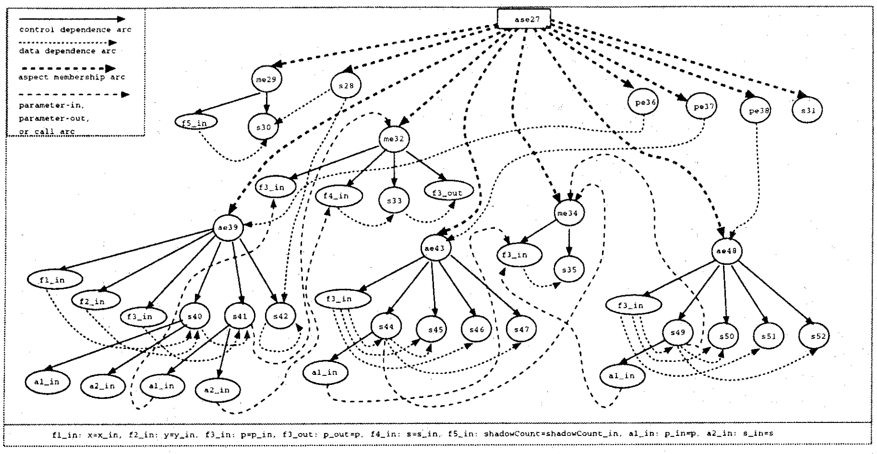

Figure 3: An ADG for aspect `PointShadowProtocol` of the program in Figure 1.

in a method dependence graph. There is an unique vertex called *introduction start vertex* to represent the entry of the introduction. To model parameter passing, we create a formal-in or formal-out vertex at each introduction start vertex (i.e., there is a formal-in vertex for each formal parameter of the introduction and there is a formal-out vertex for each formal parameter that may be modified by the introduction), and an actual-in or actual-out vertex at each call site in the introduction (i.e., there is an actual-in vertex for each actual parameter of the introduction, and an actual-out vertex for each actual parameter that may be modified by the introduction). We also create a call vertex at each call site of the introduction. Moreover, each actual parameter vertex is control dependent on the call vertex of the introduction, and each formal parameter vertex is control dependent on the introduction start vertex. Finally, we treat instance variables at the introduction entry and call sites in the introduction as parameters. Therefore, we should also create actual-in, actual-out, formal-in, formal-out vertices for these variables.

The Aspect Dependence Graph

In order to efficiently perform analysis on an individual aspect, we present the aspect dependence graph to represent dependence relationships and the parameter passing between advice, or between advice and method in the aspect.

An *aspect dependence graph* (ADG) consists of a collection of advice, introduction, or method dependence graphs and some additional arcs and vertices. Each advice, introduction, or method dependence graph represents advice, introduction, or a method in the aspect. In an ADG, there is an *aspect start vertex* to represent the entry of the aspect. The aspect start vertex is connected to the *advice start vertex* of advice dependence graph, the *introduction start vertex* of each introduction dependence graph, and the *method start vertex* of each method dependence graph in the aspect by *aspect membership arcs*. If there is a call in the advice, introduction, or method to call another advice, introduction, or method in the aspect, we connect the advice, introduction, or method dependence graphs of the advice, introduction, or method at call sites. In this case, a *call arc* is added between a call vertex of the advice, introduction, or method and the start vertex of the advice, introduction, or method dependence graph of the called advice, introduction, or method, and *parameter-in* and *parameter-out* arcs are added to connect actual-in and formal-in vertices, and formal-out and actual-out vertices. Note that parameter-in and parameter-out arcs represent the parameter passing between advice, or between advice and method in the aspect. For the instance variables declared in an aspect, since they are accessible to all advice, introductions, and methods in the aspect, we create formal-in and formal-out vertices for all instance variables that are referenced in the advice, introductions, and methods.

Moreover, for each pointcut designator, we create a pointcut start vertex to represent the entry for the pointcut, and connect the aspect start vertex to each pointcut start vertex through aspect membership arcs to represent the membership relations among them.

Finally, similar to [10], we use *summary arcs* to represent the *transitive flow of dependencies* in the ADG. Transitive flow of dependence occurs between an actual-in vertex and an actual-out vertex if the value associated with the actual-in vertex affects the value associated with the actual-out vertex. The transitive flow of dependence can be caused by data dependence, control dependence, or both. A *summary* arc models the transitive flow of dependence across a procedure call.

Example. Figure 3 shows the ADG for aspect `PointShadowProcotol` of the program in Figure 1. In the figure, a rectangle represents the aspect start vertex and is labeled by the statement label related to the aspect entry. Circles represent statements in the aspect, including advice start, and are labeled with the corresponding statement number in the aspect. Ellipses in solid line represent formal and actual parameter vertices. For example, *ase*27 is the aspect start vertex, and *ae*39, *ae*43 and *ae*48 are the advice start vertices of advices `setting`, `settingX`, and `settingY`. Bold

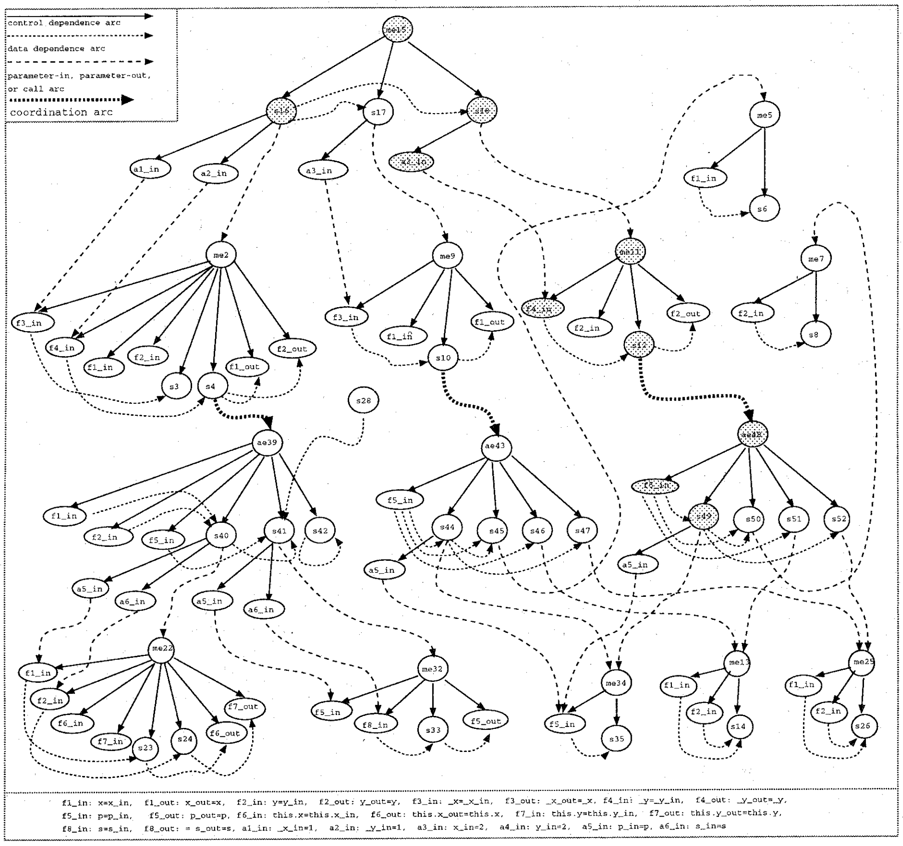
control dependence arc
data dependence arc
parameter-in, parameter-out, or call arc
coordination arc
f1_in: x=x_in, f1_out: x_out=x, f2_in: y=y_in, f2_out: y_out=y, f3_in: _x=_x_in, f3_out: _x_out=_x, f4_in: _y=_y_in, f4_out: _y_out=_y,
f5_in: p=p_in, f5_out: p_out=p, f6_in: this.x=this.x_in, f6_out: this.x_out=this.x, f7_in: this.y=this.y_in, f7_out: this.y_out=this.y,
f8_in: s=s_in, f8_out: = s_out=s, a1_in: _x_in=1, a2_in: _y_in=1, a3_in: x_in=2, a4_in: y_in=2, a5_in: p_in=p, a6_in: s_in=s

Figure 4: An ASDG of the program in Figure 1 and a slice of the program on slicing criterion (s49, p).

dashed arcs represent aspect membership arcs that connect the aspect start vertex to advice or method start vertex of advice or methods. Therefore, $(ase27, ae38)$, $(ase27, ae43)$, and $(ase27, ae48)$ are aspect membership arcs. Each advice or method start vertex is the root of a subgraph which is itself an advice or method dependence graph corresponding to the advice or method. Therefore each subgraph may contain control, data dependence arcs, parameter-in and parameter-out arcs, and summary arcs. Finally, each pointcut such as `setting`, `settingX`, or `settingY` has no corresponding subgraph since no exact body element in it, and the start vertex for each pointcut is connected to its corresponding advice start point by call arcs to represent relationship between the pointcut and advice.

2.3 Representing Interactions among Aspects and Classes

Interactions among aspects and classes can be caused from two cases: (1) Creating an object of a class from an aspect, and (2) Declaring an introduction in an aspect to add a field, method, or constructor to a class. Here we show how to represent these two cases.

Interactions by Object Creations

In AspectJ, an aspect may create an object of a class through a declaration or by using an operator such as `new`. When an aspect A creates an object of class C, there is an implicit call to C's constructor. To represent this implicit constructor call, we adds a call vertex in A at the location of object creation. A call arc connects this call vertex to C's constructor method. We also adds actual-in and actual-out vertices at the call vertex to match the formal-in and formal-out vertices in C's constructor. When there is a call site in method m_1 or advice a_1 in A to method m_2 in the public interface of C, we connect the call vertex in A to the method start vertex of m_2 to form a call arc, and also connect actual-in and formal-in vertices to form parameter-in arcs and actual-out and formal-out vertices to form parameter-out arcs. As a result, we can get a partial ASDG that represents a partial AspectJ program by connecting the ADG for A and CDG for C.

Interactions by Using Introductions

In AspectJ, an aspect A can interact with a class C by declaring an introduction I in A for adding an additional field, method, or constructor to C. To represent such an interaction, we connect the class start vertex of C's class dependence graph to the introduction start vertex of the I's introduction dependence graph by a class membership arc.

2.4 Determining Weaving-Points in Non-Aspect Code

The third part of our algorithm is to determine weaving points in the SDG at which the ADG of aspect code can be connected to the SDG of non-aspect code.

In AspectJ, join points are defined in each aspect of an aspect-oriented program with the *pointcut* designator. Pointcuts are further used in the definition of *advice* which defines code that is run when join points are reached. By carefully examining join points in the pointcuts and their associated advice, one can determine the weaving points statically in the non-aspect code to facilitate the connection of non-aspect code to the aspect code. In this paper, we use weaving vertices in the SDG to represent the weaving points in the non-aspect code which can be used to connect the SDG of non-aspect code to the ADGs of aspect code.

Example. We show how to determine the weaving point for weaving the code declared in advice `settingY` to a method in class `Point`. First, from poiotcut `settingY` declaration, we knew that the code in advice `seetingY` should be inserted into method `setY` of class `Point`. But we are still unknown the exact place where we should insert the code. By examining advice `settingY`'s declaration we further knew that this advice is *after* advice. According to the AspectJ programming guide [2]: "*after advice runs after the computation 'under the joint point' finishes, i.e., after the method body has run, and just before control is returned to the caller (p.12).*" we can know that the code declared in advice `settingY` should be inserted into the place after the last statement of method `setY`, i.e., after `y = _y`.

2.5 Weaving the SDG with ADGs

The fourth part of our algorithm is to weave the SDG and the ADGs at weaving vertices to form the ASDG. It consists of two steps: (1) Inserting weaving-vertices to the SDG to represent the corresponding weaving-points determined in the third part of our algorithm, and (2) Adding some special kinds of dependence arcs between the weaving-vertices and their corresponding advice start vertices of the advice dependence graphs in the ADG to form the ASDG.

Generally, an AspectJ program consists of classes, interfaces, and aspects. In order to execute the program, the program must include a special class called `main()` class. The program first starts the `main()` class, and then transfers the execution to other classes.

We use the ASDG to represent a complete AspectJ program. An ASDG of an aspect-oriented program consists of a collection of advice dependence graphs each representing advice of an aspect, introduction dependence graphs each representing an introduction of an aspect, method dependence graphs each representing a `main()`method or a standing method in a class or an aspect, and some additional dependence arcs to represent direct program dependencies between a call and the called advice, introduction, or method and transitive interprocedural data dependencies.

To construct the ASDG for a complete AspectJ program, we first construct the SDG for the non-aspect code and then insert the weaving vertices obtained from the third part of our algorithm to the SDG. After that, we use a *coordination arc* to connect each weaving vertex to the advice start vertex of its corresponding advice dependence graph. A call arc is added between advice, introduction, or method call vertex and the start vertex of the advice, introduction, or method dependence graph of the called advice, introduction, or method. Actual and formal parameter vertices are connected by parameter arcs. We also add the summary arcs for advice, introduction, or methods in a previously analyzed aspect between the actual-in and actual-out vertices at call sites.

Example. Figure 4 shows the complete ASDG for the program in Figure 1. The construction of the graph includes (1) the advice dependence graphs for advice `settingX`, `settingY`, and `setting`, (2) the method dependence graphs for `main()` method and standing methods `Point`, `getX`, `getY`, `setX`, `setY`, `printPosition` of class `Point`, (3) the method dependence graphs for methods `Shadow` and `printPosition` of class `Shadow`, (4) the method dependence graphs for methods `associate` and `getShadow` and the representation of introduction `Point.shadow` for asepct `PointShadowProtocol`, and (5) the connection of each subgraph using call, parameter-in and parameter-out arcs.

3 Slicing Aspect-Oriented Programs

In this section, we define some notions about static slicing of an aspect-oriented program, and show how to compute static slices of the program based on its ASDG.

In understanding and maintenance of aspect-oriented software, information that can answer the following questions may help software developers to understand a program's behavior.

(1) What code might affect by a statement in an aspect-oriented program ?

(2) What non-aspect code might affect by a statement of aspect code in an aspect-oriented program ?

(3) What aspect code might affect by a statement of non-aspect code in an aspect-oriented program ?

In order to compute slices that answer these questions, we can define three types of slicing problems.

1. A *static slicing criterion* for an aspect-oriented program is a tuple (s_1, v_1), where s_1 is a statement in the program and v_1 is a variable used at s_1, or a call called at s_1. A *static slice* $SS(s_1, v_1)$ of an aspect-oriented program on a given static slicing criterion (s_1, v_1) consists of all statements in the program that possibly affect the value of the variable v_1 at s_1 or the value returned by the call v_1 at s_1.

2. A *static slicing criterion* for an aspect-oriented program is a tuple (s_2, v_2), where s_2 is a statement in the non-aspect code of the program and v_2 is a variable used at s_2, or a call called at s_2. A *static slice* $SS(s_2, v_2)$ of an aspect-oriented program on a given static slicing criterion (s_2, v_2) consists of all statements in the aspect code that possibly affect the value of the variable v_2 at s_2 or the value returned by the call v_2 at s_2.

3. A *static slicing criterion* for an aspect-oriented program is a tuple (s_3, v_3), where s_3 is a statement in the aspect code of the program and v_2 is a variable used at s_3, or a call called at s_3. A *static slice* $SS(s_3, v_3)$ of an aspect-oriented program on a given static slicing criterion (s_3, v_3) consists of all statements in the non-aspect code of the program that possibly affect the value of the variable v_3 at s_3 or the value returned by the call v_3 at s_3.

Since the ASDG proposed for an aspect-oriented program can be regarded as an extension of the Larsen-Harrold SDG, we can use the two-pass slicing algorithm proposed in [12] to compute a static slice of an aspect-oriented program based on the ASDG.

In the first step, the algorithm traverses backward along all arcs except parameter-out arcs, and set marks to those vertices reached in the ASDG, and then in the second step, the algorithm traverses backward from all vertices having marks during the first step along all arcs except call and parameter-in arcs, and sets marks to reached vertices in the ASDG. The slice is the union of the vertices of the ASDG have marks during the first and second steps. Similar to the backward slicing described above, we can also apply the forward slicing algorithm [10] to the ASDG to compute a forward slice of an aspect-oriented program.

Example. Figure 4 shows a backward slice which is represented in shaded vertices and computed with respect to the slicing criterion (s49, p).

4 Concluding Remarks

In this paper, we presented a slicing algorithm for aspect-oriented software. To solve this problem, we developed a dependence-based representation called *aspect-oriented system dependence graph*, which extends previous dependence graphs, to represent aspect-oriented software. The aspect-oriented system dependence graph consists of three parts: (1) a system dependence graph for non-aspect code, (2) a group of aspect dependence graphs for aspect code, and (3) some additional dependence arcs used to connect the system dependence graph to the aspect dependence graphs. After that, we showed how to compute a static slice of an aspect-oriented program based on its aspect-oriented system dependence graph. We believe that in addition to computing static slices of an aspect-oriented program, the aspect-oriented system dependence graph can also be used as an underlying base to develop software engineering tools for testing and debugging aspect-oriented software [20].

While our initial exploration used AspectJ as our target language, the concept and approach presented in this paper are language independent. However, the implementation of a slicing tool may differ from one language to another because each language has its own structure and syntax which must be handled carefully.

As one of our future researches, we plan to develop a slicing tool for AspectJ which includes a generator for automatically constructing the aspect-oriented system dependence graph for an AspectJ program and a slicer for computing static slices of an AspectJ program.

References

[1] H. Agrawal, R. Demillo, and E. Spafford, "Debugging with Dynamic Slicing and Backtracking," *Software-Practice and Experience*, Vol.23, No.6, pp.589-616, 1993.

[2] The AspectJ Team, "The AspectJ Programming Guide," 2001.

[3] S. Bates, S. Horwitz, "Incremental Program Testing Using Program Dependence Graphs," *Conference Record of the 20th Annual ACM SIGPLAN-SIGACT Symposium of Principles of Programming Languages*, pp.384-396, Charleston, South California, ACM Press, 1993.

[4] J. Beck and D. Eichmann, "Program and Interface Slicing for Reverse Engineering," *Proceeding of the 15th International Conference on Software Engineering*, pp.509-518, Baltimore, Maryland, IEEE Computer Society Press, 1993.

[5] D. W. Binkley and K. B. Gallagher, "Program slicing," *Advanced in Computer Science*, 1996.

[6] J. L. Chen, F. J. Wang, and Y. L. Chen, "Slicing Object-Oriented Programs," *Proceedings of the APSEC'97*, pp.395-404, Hongkong, China, December 1997.

[7] A. De Lucia, A. R. Fasolino, and M. Munro, "Understanding function behaviors through program slicing," *Proceedings of the Fourth Workshop on Program Comprehension*, Berlin, Germany, March 1996.

[8] J.Ferrante, K.J.Ottenstein, J.D.Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Transaction on Programming Language and System*, Vol.9, No.3, pp.319-349, 1987.

[9] K. B. Gallagher and J. R. Lyle, "Using Program Slicing in Software Maintenance,"*IEEE Transaction on Software Engineering*, Vol.17, No.8, pp.751-761, 1991.

[10] S. Horwitz, T. Reps and D. Binkley, "Interprocedural Slicing Using Dependence Graphs,"*ACM Transaction on Programming Language and System*, Vol.12, No.1, pp.26-60, 1990.

[11] 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.

[12] L. D. Larsen and M. J. Harrold, "Slicing Object-Oriented Software," *Proceeding of the 18th International Conference on Software Engineering*, German, March, 1996.

[13] J. Q. Ning, A. Engberts, and W. Kozaczynski, "Automated Support for Legacy Code Understanding," *Communications of ACM*, Vol.37, No.5, pp.50-57, May 1994.

[14] K. J. Ottenstein and L. M. Ottenstein, "The Program Dependence Graph in a software Development Environment," *ACM Software Engineering Notes*, Vol.9, No.3, pp.177-184, 1984.

[15] F. Tip, "A Survey of Program Slicing Techniques," *Journal of Programming Languages*, Vol.3, No.3, pp.121-189, September, 1995.

[16] F. Tip, J. D. Choi, J. Field, and G. Ramalingam "Slicing class hierarchies in C++," *Proceedings of the 11th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp.179-197, October, 1996.

[17] M. Weiser, "Program Slicing," *IEEE Transaction on Software Engineering*, Vol.10, No.4, pp.352-357, 1984.

[18] J. Zhao, "Applying Program Dependence Analysis to Java Software," *Proc. Workshop on Software Engineering and Database Systems, 1998 International Computer Symposium*, pp.162-169, December 1998.

[19] J. Zhao, "Slicing Concurrent Java Programs," *Proc. Seventh IEEE International Workshop on Program Comprehension*, pp.126-133, May 1999.

[20] J. Zhao, "Dependence Analysis of Aspect-Oriented Software and Its Applications to Slicing, Testing, and Debugging," Technical Report SE-2001-134-17, Information Processing Society of Japan (IPSJ), October 2001.