Slicing Aspect-Oriented Software

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

Program slicing has many applications in software engineering activities including program comprehension, debugging, testing, maintenance, and model checking. 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: a system dependence graph for non-aspect code, a group of dependence graphs for aspect code, and some additional dependence arcs used to connect the system dependence graph to the dependence graphs for aspect code. After that, we show how to compute a static slice of an aspect-oriented program based on the ASDG.

```
 1 begin                       1  begin
 2   read(X,Y);                2    read(X,Y);
 3   Total := 0.0;             3    Total := 0.0;
 4   Sum := Y;                 4    ▭
 5   if X <= 1 then            5    if X <= 1 then
 6     Sum := Y;               6      ▭
 7     else                    7      else
 8       begin                 8        begin
 9         read(Z);            9          ▭
10         Total := X * Y;    10          Total := X * Y;
11       end;                 11        end;
12   end if                   12    end if
13   Write(Total, sum);       13    ▭
14 end                        14  end
      (a)                                (b)
```

Figure 1. A program fragment and its slice on criterion (Total,14).

1. Introduction

Program slicing, originally introduced by Weiser [21], 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 [11, 13, 14, 21]. In such languages, slicing is typically performed by using a control flow graph or a dependence graph [5, 19]. Program slicing has many applications in software engineering activities including program understanding [9], debugging [1], testing [4], maintenance [12], and model checking [10]. Recently program slicing has been also applied to object-oriented software to handle various object-oriented features [6, 18, 20, 24] as well as software architecture [22].

To understand the basic idea of program slicing, consider a simple example in Figure 1 that shows: (a) a program fragment and (b) its slice with respect to the slicing criterion (`Total`,14). The resulted slice consists of only those statements in the program that might affect the value of variable `Total` at line 14. The lines represented by small rectangles are statements that have been sliced away.

Aspect-oriented programming has been proposed as a technique for improving separation of concerns in software design and implementation [15]. Aspect-oriented programming works by providing explicit mechanisms for capturing the structure of crosscutting concerns in software systems.

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 join point, 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 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 slicing approaches have been proposed for procedural or object-oriented software, there is no slicing algorithm for aspect-oriented software until now, and due to some specific features in aspect-oriented software, existing slicing algorithms for procedural or object-oriented software can not be applied to aspect-oriented software straightforwardly.

In this paper, we propose an approach to slicing aspect-oriented software. To this end, we develop a dependence-based representation called *aspect-oriented system dependence graph*, that extends previous dependence graphs, to

Table 1. Advice in AspectJ [3].

Type of advice	Points in the program execution
`before(Formals) :`	execute before the join point
`after(Formals) returning [ (Formal) ] :`	execute after the join point if it returns normally. The optional formal gives access to the returned value.
`after(Formals) throwing [ (Formal) ] :`	execute after the join point if it throws an exception. The optional formal gives access to the `Throwable` exception value.
`after(Formals) :`	execute after the join point both when it returns normally and when it throws an exception.
`Type around(Formals) [ throws TypeList] :`	execute instead of the join point. The join point can be executed by calling `proceed`.

represent aspect-oriented software. The aspect-oriented system dependence graph consists of three parts: a system dependence graph for non-aspect code, a group of dependence graphs for aspect code, and some additional dependence arcs used to connect the system dependence graph to the dependence graphs for aspect code. 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 is a new dependence-based representation 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 gives some background information related to this research. Section 3 presents the aspect-oriented system dependence graph for aspect-oriented software. Section 4 shows how to compute static slices based on the graph. Section 5 discusses some related work. Concluding remarks are given in Section 6.

2 Background

2.1 AspectJ

We assume that readers are familiar with the basic concepts of aspect-oriented programming and design, and in this paper, we use AspectJ [3] as our target language to show the basic idea of our slicing approach. The selection of AspectJ is based on that it is one of most popular aspect-oriented language in the community.

Below, we use a sample program taken from [3] to briefly introduce the AspectJ. The program shown in Figure 2 associates shadow points with every `Point` object and contains one `PointShadowProtocol` aspect that stores a shadow object in every `Point` and two classes `Point` class and `Shadow`.

AspectJ [3] 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, pointcut, advice, introduction, and aspect.

The *join point* is an 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 points in AspectJ are well-defined points in the execution of a program. The join points in AspectJ are *method* or *constructor call*, *method* or *constructor execution*, *class* or *object initialization*, *field reference* or *assignment*, and *handler execution* [3].

A *pointcut* is a set of join points that optionally exposes some of the values in the execution of those join points. AspectJ defines several primitive *pointcut designators* that can identify all types of join points. Pointcuts in AspectJ can be composed and new pointcut designators can be defined according to these combinations. For example, In aspect `PointShadowProtocol` three pointcuts are declared with the names of `setting`, `settingX`, and `settingY`.

Advice is used to define some code that is executed when a pointcut is reached. ApsectJ provides three types of advice, that is, *before*, *after*, and *around*. In addition, there are also two special cases of *after* advice, called *after returning* and *after throwing*. For example, In aspect `PointShadowProtocol`, there are three *after* advice `setting`, `settingX`, and `settingY`. Table 1 shows all types of advice in AspectJ.

Advice declarations can change the behavior of classes they crosscut, but can not change their static type structure. For crosscutting concerns that can operate over the static structure of type hierarchies, AspectJ provides forms of introduction.

Introduction in AspectJ can be used by an aspect to add new fields, constructors, or methods (even with bodies) into given interfaces or classes. Introduction can be public or private, where a private introduction means only code in the aspect that declared it can refer or access the introduced fields, constructors, or methods. For example, In aspect `PointShadowProtocol`, introduction declaration `private Shadow Point.shadow;` privately introduces a field named `shadow` of type `Shadow` in `Point`. This means that only code in the aspect can refer to `Point`'s `shadow` field.

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 advice, pointcut, and introduction declarations as well as other declarations such as method declarations, that are permitted in class declarations. For example, the program in Figure 2 defines one aspect named

```
 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 = getShadow(p);
  s50     s.y = p.getY() + Shadow.offset;
  s51     p.printPosition();
  s52     s.printPosition();
       }
     }
```

Figure 2. A sample AspectJ program.

`PointShadowProtocol`.

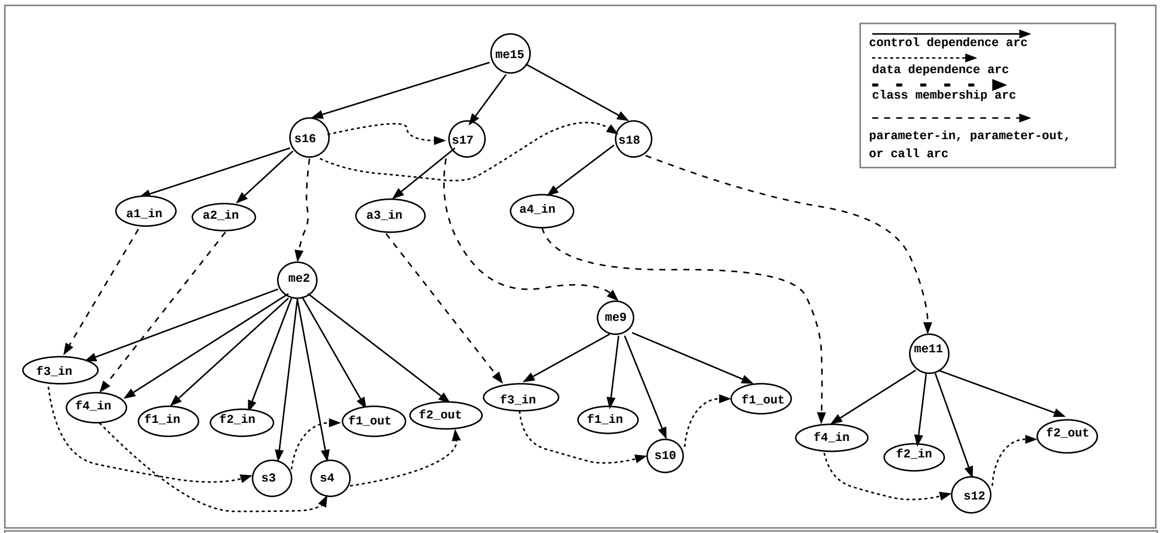
An AspectJ program can be divided into two parts: *non-aspect code* which includes some classes, interfaces, and other language constructs as in Java, and *aspect code* which includes aspects for modeling crosscutting concerns in the program. Moreover, any implementation of AspectJ is to ensure that the aspect and non-aspect code run together in a properly coordinated 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 [3].

2.2 The System Dependence Graph for Object-Oriented Software

The *system dependence graph* (SDG) [18, 23] of an object-oriented program 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 or indirect dependencies between a call and the called method and transitive interprocedural data dependencies.

The *method dependence graph* (MDG) of a method is a digraph whose vertices represent statements or predicate expressions in the method and arcs represent two types of dependence relationships, i.e., *control dependence*, and *data dependence*. Control dependence represents control conditions on which the execution of a statement or expression depends in the method. Data dependence represents the data flows between statements in the method. Each MDG has a unique vertex called *method start vertex* to represent the entry of the method.

To model parameter passing, formal parameter vertices are created to associate with each method start vertex. 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. Also, a *call vertex* and actual parameter vertices are created to associate with each call site. There is an *actual-in vertex* for each actual parameter and an *actual-out vertex* for each actual param-



```
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,  a1_in: _x_in=1,
a2_in: _y_in=1,  a3_in: _x_in=2,  a4_in: _y_in=2
```

Figure 3. A SDG for non-aspect code of the program in Figure 2.

eter that may be modified by the called method. In addition, each formal parameter vertex is control dependent on the method start vertex, and each actual parameter vertex is control dependent on the call vertex.

The construction of the complete SDG can be performed by connecting MDGs at call sites. A *call arc* which represents the call relationships is added between the call vertex of the calling method's MDG and the start vertex of the called method's MDG. 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*. These parameter arcs can represent parameter passing. Moreover, to represent the *transitive flow of dependencies* in the SDG, *summary arcs* [14] are created by connecting an actual-in vertex to an actual-out vertex if the value associated with the actual-in vertex affects the value associated with the actual-out vertex.

Example. Figure 3 shows the SDG for non-aspect code of the program in Figure 2.

3 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 join points, advice, aspects, and their associated constructs. These aspect-oriented features may have an impact on the development of the 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) an SDG for non-aspect code, (2) a group of dependence graphs for aspect code, and (3) some additional dependence arcs used to connect the SDG and the dependence graphs for aspect code.

Our construction algorithm for the ASDG of an aspect-oriented program consists of four steps:

(1) Constructing the SDG for non-aspect code of the program, by using existing techniques introduced in Section 2 for object-oriented software.

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

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

(4) Weaving the SDG and the advice dependence graphs 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.

Note that in order to focus on the key ideas of our approach to slicing aspect-oriented software, here we will not discuss the first part of our construction algorithm. The SDG for non-aspect code can be constructed by using the techniques introduced in Section 2. In the rest of this section, we describe the rest three parts of our construction algorithm in more detail.

3.1 Representing Aspect Code

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 aspects may include advice,

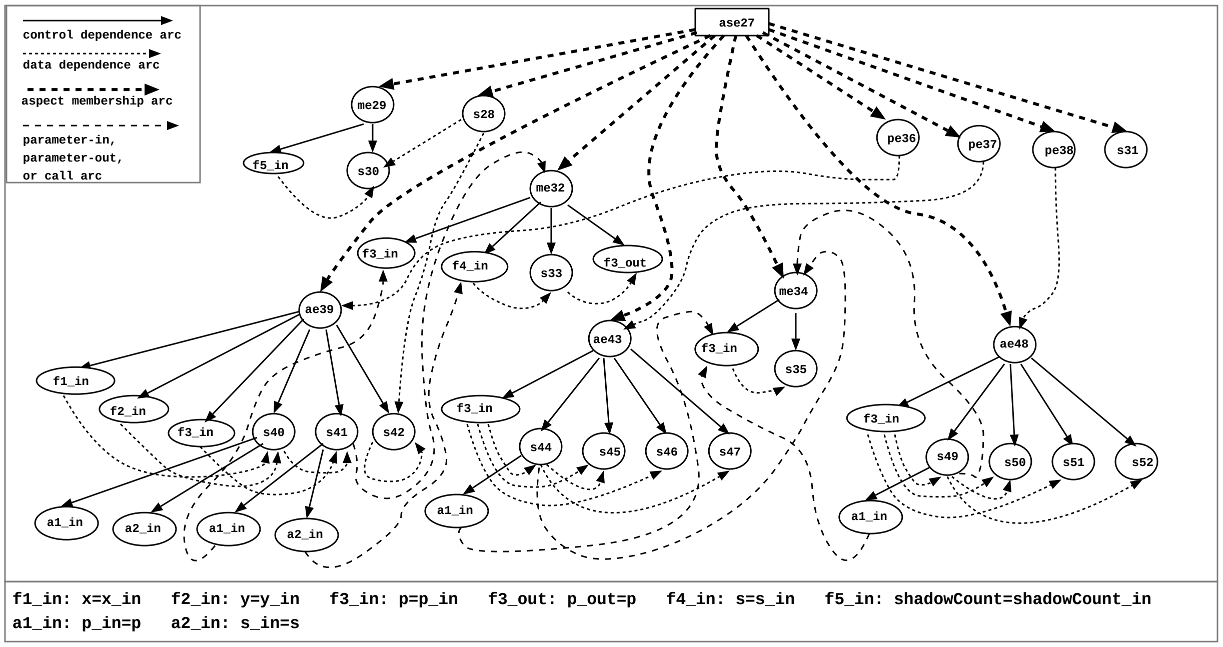


Figure 4. An AsDG for aspect PointShadowProtocol of the program in Figure 2.

making AspectJ be able to define crosscutting effects. The declaration of those effects is localized. In the following, we show how to represent advice, introductions, and aspects by using dependence graphs.

3.1.1 Advice

We use a dependence graph called *advice dependence graph* (ADG) to represent advice in an aspect. The ADG of advice is similar to the MDG of a method such that its vertices represent statements or predicate expressions in the advice, and its arcs represent control or data dependencies between vertices. Each advice has a unique vertex called *advice start vertex* to represent the entry of the advice.

3.1.2 Introductions

We use a dependence graph called *introduction dependence graph* (IDG) to represent an introduction in an aspect. The IDG of an introduction is similar to the MDG of a method 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 dependencies between these statements. There is a unique vertex called *introduction start vertex* in the IDG to represent the entry of the introduction.

3.1.3 Pointcuts

Pointcuts declared in an aspect contain no body. Therefore, for each pointcut designator, we only create a *pointcut start vertex* to represent the entry into the pointcut.

3.1.4 Aspects

In order to efficiently perform analysis on an individual aspect, we use a dependence graph called *aspect dependence graph* to represent a single aspect in an aspect-oriented program.

The *aspect dependence graph* (AsDG) of an aspect is a digraph that consists of a number of ADGs, IDGs, and MDGs each representing advice, an introduction, or a method in the aspect, and some special kinds of dependence arcs to represent direct or indirect dependencies between a call and the called advice, introduction, or method and transitive interprocedural data dependencies in the aspect. Each AsDG has a unique vertex called *aspect start vertex* to represent the entry into the aspect. The aspect start vertex is connected to each start vertex of an ADG, IDG, or MDG in the aspect by *aspect membership arcs* to represent the membership relations.

We use the techniques introduced in Section 2 to model parameter passing in an aspect. Formal-in and formal-out vertices are associated with each advice, introduction, or method start vertex, and actual-in and actual-out vertices are associated with each call vertex representing a call site in the aspect. Each formal parameter vertex is control-dependent on the start vertex, and each actual parameter vertex is control-dependent on the call vertex.

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.

Finally, for each pointcut, we connect the aspect start vertex to each pointcut start vertex through aspect membership arcs, and also connect each pointcut start vertex to its

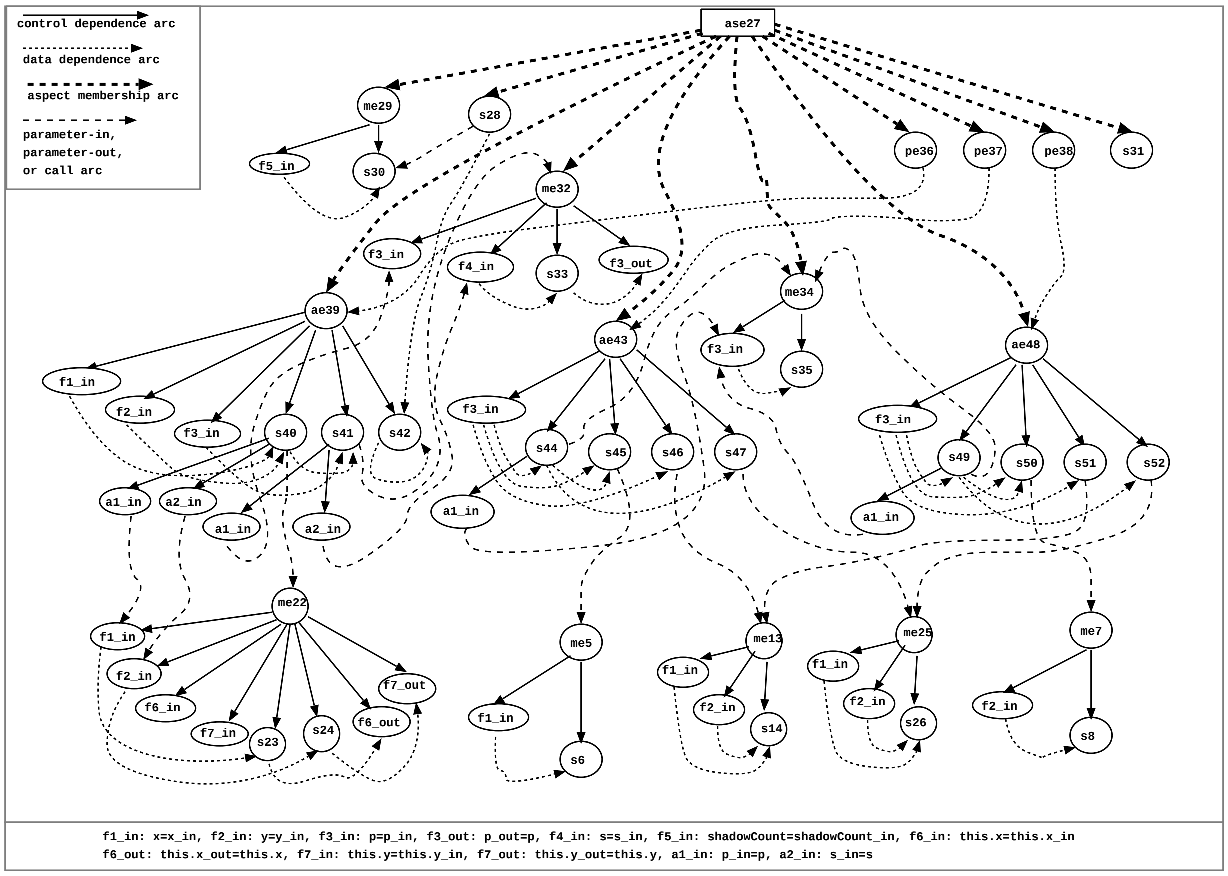

control dependence arc
data dependence arc
aspect membership arc
parameter-in, parameter-out, or call arc
ase27
me29
s28
f5_in
s30
me32
pe36
pe37
pe38
s31
f3_in
f4_in
s33
f3_out
me34
ae39
ae43
s35
ae48
f1_in
f2_in
f3_in
s40
s41
s42
s44
s45
s46
s47
s49
s50
s51
s52
a1_in
a2_in
a1_in
a2_in
a1_in
a1_in
me22
me5
me13
me25
me7
f6_in
f7_in
s23
s24
f6_out
f7_out
s6
s14
s26
s8
f1_in: x=x_in, f2_in: y=y_in, f3_in: p=p_in, f3_out: p_out=p, f4_in: s=s_in, f5_in: shadowCount=shadowCount_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: p_in=p, a2_in: s_in=s

Figure 5. A partial ASDG for representing interactions between aspect PointShadowProtocol and classes Point and Shadow.

corresponding advice start vertex by call arcs to represent relationship between them.

Example. Figure 4 shows the AsDG of aspect `PointShadowProtocol`. For example, in the figure, $ase27$ is the aspect start vertex, and $ae39$, $pe36$, $ie31$ and $me32$ are advice, pointcut, introduction, and method start vertices respectively. $(ase27, ae39)$, $(ase27, pe36)$, $(ase27, ie31)$, and $(ase27, me32)$ are aspect membership arcs. Moreover, each advice, introduction, or method start vertex is the root of a subgraph which is itself an ADG, IDG, or MDG.

3.2 Representing Interactions between 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 a public introduction in an aspect to add a field, method, or constructor to a class. Here we show how to represent these two cases.

3.2.1 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 add a call vertex in A at the location of object creation. A call arc connects this call vertex to C's constructor MDG . We also add actual-in and actual-out vertices at the call vertex to match the formal-in and formal-out vertices in C's constructor MDG. 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 AsDG for A and the class dependence graph for C. For example,

3.2.2 Interactions by Using Introductions

In AspectJ, an aspect A can also interact with a class C by declaring a public introduction I in A for adding an additional field, method, or constructor to C. To represent such

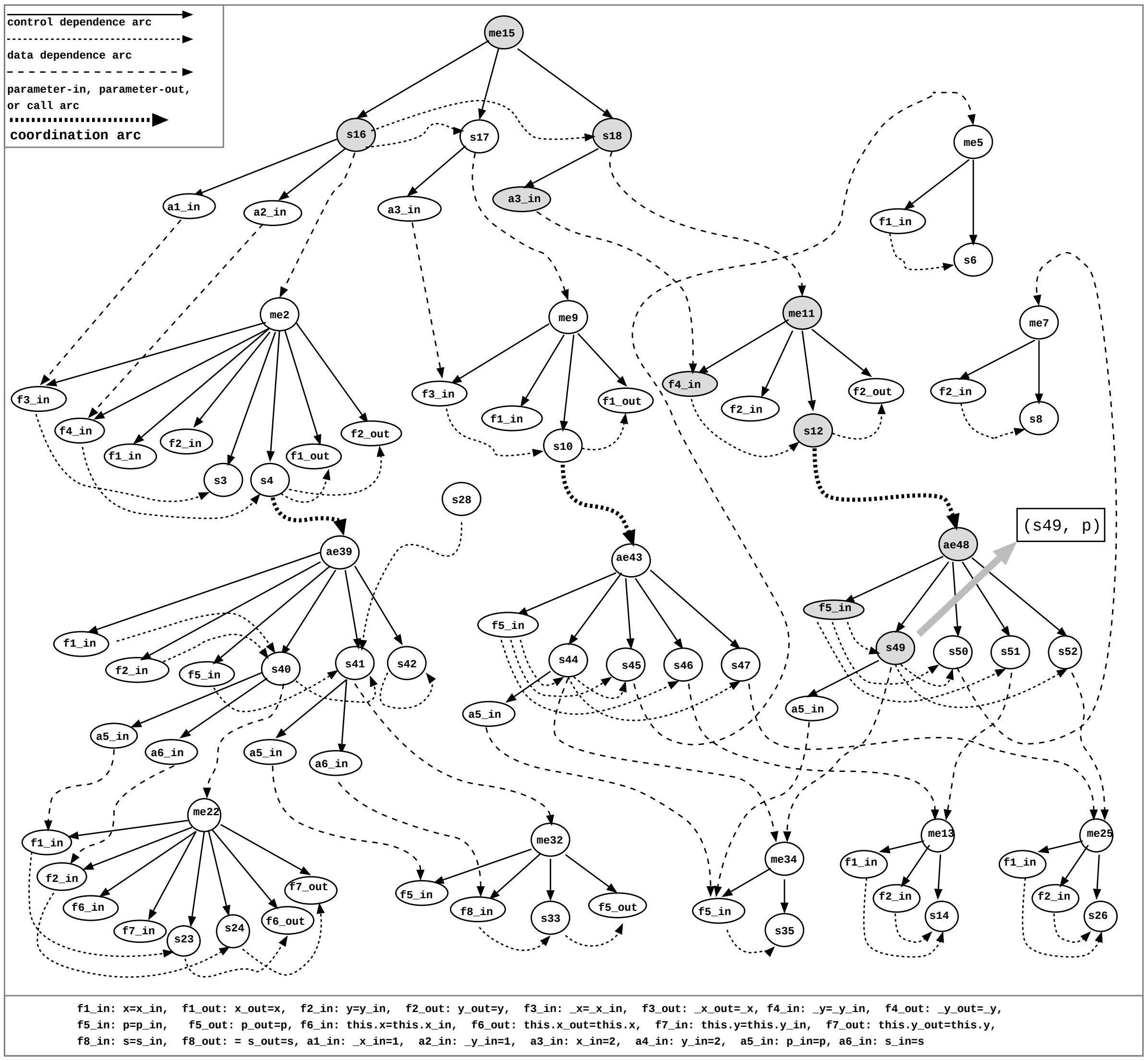

control dependence arc
data dependence arc
parameter-in, parameter-out, or call arc
coordination arc
(s49, p)
f1_in: x=x_in, f1_out: x_out=x, f2_in: y=y_in, f2_out: y_out=y, f3_in: _x=_x_in, f3_out: _x_out=_x, f4_in: _y=_y_in, f4_out: _y_out=_y,
f5_in: p=p_in, f5_out: p_out=p, f6_in: this.x=this.x_in, 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 6. An ASDG of the program in Figure 2 and a slice of the program on slicing criterion (s49, p).

an interaction, we connect the class start vertex of C's class dependence graph to the introduction start vertex of the I's IDG by a class membership arc.

Example. Figure 5 shows an AsDG for representing the interactions among aspect `PointShadowProtocol` and two classes `Point` and `Shadow`.

3.3 Determining Weaving Points in 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*. By carefully examining join points declared in the pointcuts and their associated advice, one can determine the weaving points statically in the non-aspect code to facilitate the connection of the 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 AsDGs of aspect code.

For example, in order to determine the weaving point for weaving the code declared in advice `settingY` to a method in class `Point`. First, from pointcut `settingY` declaration, we knew that the code in advice `settingY` should be inserted into method `setY` of class `Point`. However, we still do not know the exact place where we should insert the code. By examining advice `settingY`'s declaration we further know that this advice is *after* advice. According to the AspectJ programming guide [3]: "*after advice runs after the computation 'under the join point' finishes, i.e., after the method body has run, and just before control is returned to the caller (p.12),*" we know that the

Table 2. Parameters which contribute to the size of an ASDG.

Vertices	*Large number of statements in a single advice, introduction, or method*
Arcs	*Large number of arcs in a single advice, introduction, or method*
Params	*Largest number of formal parameters for any advice, introduction, or method*
ObjectVar	*Largest number of instance variables in an aspect or a class*
CallSites	*Largest number of call sites in any advice, introduction, or method*
TreeDepth	*Depth of inheritance tree determining number of possible indirect call destinations*
Module	*Number of advice, introductions, and methods*

code declared in advice `settingY` should be inserted into the place after the last statement of method `setY`, i.e., after `y = _y`. Similarly, we can determine other weaving points in the non-aspect code.

3.4 Weaving the SDG with ADGs to Form the Complete 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.

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 ADG. A call arc is added between a call vertex and the start vertex of the ADG, IDG, or MDG 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 6 shows the complete ASDG of the sample AspectJ program in Figure 2. The construction of the graph includes (1) the ADGs for advice `settingX`, `settingY`, and `setting`, the MDGs for methods `associate` and `getShadow` and the representation of introduction `Point.shadow` in aspect `PointShadowProtocol`, (2) the MDGs for `main()` method and standing methods `Point`, `getX`, `getY`, `setX`, `setY`, `printPosition` of class `Point`, (3) the MDGs for standing methods `Shadow` and `printPosition` of class `Shadow`, (4) the connection of each subgraph using call, parameter-in and parameter-out arcs.

3.5 Cost of Constructing the ASDG

The size of the ASDG is critical for applying it to the practical development environment for aspect-oriented software. Here we try to predict the size of the ASDG based on the work done by Larsen and Harrold [18] who gave an estimate of the size of the SDG for object-oriented software. In AspectJ, class declarations are like class declarations in Java except that they can also include pointcut declarations, and aspect declarations are like class declarations in Java except that they can also include pointcut declarations, advice declarations, introduction declarations. As a result, we can apply their approximation here to estimate the size of the ASDG of an aspect-oriented program by regarding an aspect as a class-like unit and an introduction or advice as a method-like unit. However, since our ASDG is constructed for an aspect-oriented program, there may be difference in its sizes compared to the SDG for an object-oriented programs.

Table 2 lists the variables that contribute to the size of an ASDG. We give a bound on the number of parameters for any advice, introduction, or method β (ParamVertices(β)), and use this bound to compute upper bound on the size of a single advice, introduction, or method β (Size(β)). Based on the Size(β) and the number of advice, introductions, and methods in an aspect-oriented program Modules, we can compute the upper bound on the number of vertices in an ASDG including all aspects and classes Size(ASDG) that contribute to the size of the program.

- ParamVertices(m) = Params+ObjectVar
- Size(β) = O(Vertices*CallSites*(1+TreeDepth*(2*ParamVertices))+2*ParamVertices)
- Size(ASDG) = O(Size(β) * Modules)

Note that Size(ASDG) provides only a rough upper bound on the number of vertices in an ASDG. In practice an ASDG may be considerably more space efficient.

4 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.

- Which statements might affect a statement in an aspect-oriented program ?
- Which statements in non-aspect code might affect a statement of aspect code in an aspect-oriented program ?
- Which statements in aspect code might affect a statement of non-aspect code in an aspect-oriented program ?

In order to answer these questions, we can define three types of static slices for an aspect-oriented program.

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

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

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

Note that according to the above definitions, slice(1) can be considered as a superset of slice(2) and slice(3), and slice(2) and slice(3) have no shared elements.

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 [14] 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 marked during the first and second steps. Similar to the backward slicing described above, we can also apply the forward slicing algorithm [14] to the ASDG to compute a forward slice of an aspect-oriented program.

Example. Figure 6 shows a backward slice which is represented in shaded vertices and computed with respect to the slicing criterion `(s49, p)`.

5 Related Work

As an essential analysis technique useful for many software engineering tasks, program slicing has been widely studied in the literature. In this section, we review some related work on program slicing that directly or indirectly influences our work on slicing aspect-oriented software. To the best of our knowledge, it is the first time to perform slicing on aspect-oriented software.

5.1 Static Slicing of Procedural Software

Ferrante *et al.* [11] presented the *program dependence graph* (PDG) to explicitly represent control and data dependences in a sequential procedural program with single procedure. Horwitz *et al.* [14] extended the PDG to introduce the *system dependence graph* (Horwitz-Reps-Binkley SDG) to represent a sequential procedural program with multiple procedures. Based on these dependence graphs, one can perform intraprocedural or interprocedural slicing on procedural programs. Cheng [8] presented the *process dependence net* (PDN) to slice concurrent procedural programs. The PDN is the generalization of the PDG to represent program dependencies in a concurrent procedural program with single procedure. Krinke [16] also considered to slice concurrent programs by using the *threaded program dependence graph*. Recently, Harman and Danicic [13] proposed a new slicing technique called *amorphous slicing* to compute program slices of procedural software. However, slicing techniques for procedural languages lack the ability to represent aspect-oriented features in aspect-oriented software. For more detailed information on slicing of procedural languages, one can refer to [5, 19].

5.2 Static Slicing of Object-Oriented Software

Larsen and Harrold [18] proposed a static slicing algorithm for sequential object-oriented programs by extending Horwitz-Reps-Binkley SDG to represent object-oriented programs. Their SDGs can be used to represent many object-oriented features such as classes and objects, polymorphism, and dynamic binding. Since the SDGs they computed belong to a class of Horwitz-Reps-Binkley SDG, they can use the two-pass slicing algorithm introduced in [14] to compute static slices of object-oriented software. Krishnaswamy [17] proposed another approach to slicing object-oriented software by using the *object-oriented program dependency graph*(OPDG). His approach can compute polymorphic slices of an object-oriented program based on the OPDG. Chen *et al.* [6] also extended the PDG to the *object-oriented dependency graph* for modeling object-oriented software and computed static slices for sequential object-oriented programs. Slicing concurrent object-oriented programs has also been considered. Zhao [24] presented a dependence graph-based approach to slicing concurrent object-oriented programming languages such as Java. Chen *et al.* [7] proposed an approach to slicing Tagged Objects in Ada 95 programs. Although these slicing algorithms can handle many features of object-oriented software, they generally lack the ability to represent specific aspect-oriented features such as join point, pointcut, advice, and aspect in aspect-oriented software.

6 Concluding Remarks

In this paper, we proposed an approach to slicing 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 dependence graphs for aspect code, and (3) some additional dependence arcs used to connect the system dependence graph to the dependence graphs for aspect code. 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 [25].

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 research avenues, 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] AspectJ home page: `http://www.aspectj.org`.

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

[4] 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.

[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] Z. Chen, B. Xu, and H. Yang, "Slicing Tagged Objects in Ada 95," *Proceedings of the Ada-Europe'2001*, LNCS, vol. 2043, pp.100-112, Springer-Verlag, 2001.

[8] J. Cheng, "Process Dependence Net of Distributed Programs and Its Applications in Development of Distributed Systems," *Proceedings of the IEEE-CS 17th Annual COMPSAC*, pp.231-240, U.S.A., 1993.

[9] 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.

[10] Matthew B. Dwyer and John Hatcliff, "Slicing Software For Model Construction," *Proc. ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99)*, January 1999.

[11] 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.

[12] 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.

[13] Mark Harman and Sebastian Danicic, "Amorphous Program Slicing," *Proc. IEEE International Workshop on Program Comprehension (IWPC'97)*, pp.70-79, Dearborn, Michigan, May 1997.

[14] 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.

[15] 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*, pp.220-242, LNCS, Vol.1241, Springer-Verlag, June 1997.

[16] J. Krinke, "Static Slicing of Threaded Programs," *proc. ACM SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pp.35-42, ACM Press, 1998.

[17] A. Krishnaswamy, "Program Slicing: An Application of Object-Oriented Program Dependency Graphs," Technical Report TR94-108, Department of Computer Science, Clemson University, 1994.

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

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

[20] 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.

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

[22] J. Zhao, "Applying Slicing Technique to Software Architectures," *Proc. 4th IEEE International Conference on Engineering of Complex Computer Systems*, pp.87-98, August 1998.

[23] 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.

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

[25] J. Zhao, "Dependence Analysis of Aspect-Oriented Software and Its Applications to Slicing, Testing, and Debugging," Technical Report SE-2001-135-17, Information Processing Society of Japan (IPSJ), October 2001.