System Dependence Graph Construction for Aspect-Oriented Programs

Jianjun Zhao* and Martin Rinard
Laboratory for Computer Science
Massachusetts Institute of Technology
200 Technology Square, Cambridge, MA 02139, USA
{jianjun,rinard}@cag.lcs.mit.edu

ABSTRACT

We extend previous dependence-based representations called *system dependence graphs* (SDGs) to represent aspect-oriented programs and present an SDG construction algorithm. This algorithm first constructs a *module dependence graph* (MDG) for each piece of advice, introduction, and method in aspects and classes. It then uses existing techniques to connect the MDGs at call sites to form a partial SDG. Finally, it weaves the MDG for each piece of advice into the partial SDG for those methods whose behavior may be affected by the advice. The result is the complete SDG. Our SDGs capture the additional structure present in many aspect-oriented features such as join points, advice, introduction, aspects, and aspect inheritance, and various types of interactions between aspects and classes. They also correctly reflect the semantics of aspect-oriented concepts such as advice precedence, introduction scope, and aspect weaving. SDGs therefore provide a solid foundation for the further analysis of aspect-oriented programs.

1. INTRODUCTION

The *program dependence graph* (PDG) effectively supports powerful program analysis by explicitly capturing dependence information between different program elements [6]. Although originally proposed for compiler optimizations, the PDG has also been used as an effective foundation for various software engineering tasks such as program understanding, debugging, testing, and maintenance [3, 16, 17]. PDGs were originally constructed for procedural programs [6, 7], and have been recently extended to support various object-oriented features such as classes and objects, inheritance, polymorphism, and dynamic binding [11, 13, 14, 15].

Aspect-oriented programming (AOP) [8, 19] is a new language paradigm proposed for cleanly modularizing the cross-cutting structure of concerns such as exception handling, synchronization, and resource sharing. Expressing such cross-cutting concerns using standard language constructs usually produces tangled, poorly structured code. AOP can control such code tangling, improving the structure of the program and making it easier to develop and maintain.

Aspect-oriented languages present unique opportunities and problems for program representation schemes. To enable the development of effective analysis for aspect-oriented programs, the program representation scheme must appropriately preserve the structure associated with the use of features such as joint points, advice, and introduction. It is therefore of interest to develop appropriate program representations for aspect-oriented programs.

Although researchers have developed many representations for procedural and object-oriented programs, little work has been carried out on representations that preserve the structure of aspect-oriented programs. Due to some specific aspect-oriented features presented in recent AOP languages, existing representations for procedural and object-oriented programs can not be used directly for aspect-oriented programs.

In this paper we extend previous dependence-based representations called *system dependence graphs* (SDGs) to represent aspect-oriented programs and present an SDG construction algorithm. This algorithm first constructs a *module dependence graph* (MDG) for each piece of advice, introduction, and method in aspects and classes. It then uses existing techniques to connect the MDGs at call sites to form a partial SDG. Finally, it weaves the MDG for each piece of advice into the partial SDG for those methods whose behavior may be affected by the advice. The result is the complete SDG. Our SDGs capture the additional structure present in many aspect-oriented features such as join points, advice, introduction, aspects, and aspect inheritance, and various types of interactions between aspects and classes. They also correctly reflect the semantics of aspect-oriented concepts such as advice precedence, introduction scope, and aspect weaving. SDGs therefore provide a solid foundation for the further analysis of aspect-oriented programs. One key point for representing aspect weaving is to determine, for each pointcut *pc*, a set of methods in classes that might be affected by *pc*. Based on this kind of information, we can weave the MDGs for advice into the partial SDG for base code in a natural way.

Our main contribution in this paper is a new representation for aspect-oriented programs. This representation can serve as an effective foundation for program analysis. Because the representation preserves the additional structure present in

*Jianjun Zhao is on sabbatical from the 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.

```
ce0 public class Point {
 s1   protected int x, y;
me2   public Point(int x1, int y1) {
 s3     x = x1;
 s4     y = y1;
      }
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+")");
      }
    }
                    (a)
```

```
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;
        }

 p36    pointcut setting(int x, int y, Point p):
          args(x,y) && call(Point.new(int,int));
 p37    pointcut settingX(Point p):
          target(p) && call(void Point.setX(int));
 p38    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();
      }
    }                   (b)
```

Figure 1: An AspectJ program which contains (a) base code and (b) aspect code.

the aspect-oriented program, it is especially appropriate for developing software engineering tools.

The rest of the paper is organized as follows. Section 2 briefly introduces the system dependence graph for standard object-oriented programs and presents the aspect-oriented programming language AspectJ. Section 3 presents the SDG for aspect-oriented programs and the construction algorithm for an SDG. Section 4 discusses some related work; we conclude in Section 5.

2. BACKGROUND

We next briefly introduce the AspectJ language and the system dependence graphs for object-oriented programs.

2.1 Aspect-Oriented Programming with AspectJ

We present our extended SDG in the context of AspectJ, the most widely used aspect-oriented programming language. [9]. Our basic techniques, however, deal with the basic concepts of aspect-oriented programming and therefore apply to the general class of AOP languages.

AspectJ [9] is a seamless aspect-oriented extension to Java; AspectJ adds some new concepts and associated constructs to Java. These concepts and associated constructs are called join points, pointcut, advice, introduction, and aspect. We briefly introduce each of these constructs as follows.

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

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

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

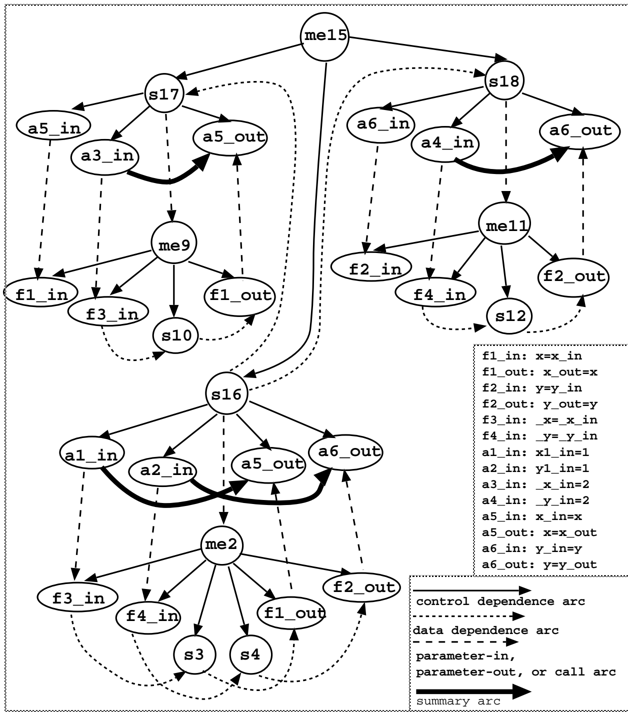

Figure 2: SDG for the base code of program in Figure 1.

replaced. For example, the aspect `PointShadowProtocol` in Figure 1 declares three pieces of after advice at `ae39`, `ae43`, and `ae48`; each is attached to the corresponding pointcut `setting`, `settingX`, or `settingY`.

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

To focus on the key ideas of our work, we do not discuss how to represent standard constructs such as classes and interfaces in this paper because they can be represented using existing techniques [13, 14]. To simplify the presentation we present only those join points related to method and constructor calls.

2.2 System Dependence Graphs for Object-Oriented Programs

The *system dependence graph* (SDG) [13, 14] for an object-oriented program is a collection of method dependence graphs; each of which represents a `main()` method or a method in a class of the program. Additional arcs represent direct or indirect dependences between a call site and the invoked method and transitive interprocedural data dependences. A *method dependence graph* (MDG) for a method m is a directed graph[1] whose vertices represent statements or predicate expressions in m; arcs represent *control* and *data dependences*. The MDG also has a unique vertex called *method entry vertex* to represent the entry into m.

Formal parameter vertices are used to model parameter passing between methods. 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. Each call site has an associated *call vertex* and actual parameter vertices: an *actual-in vertex* for each actual parameter and an *actual-out vertex* for each actual parameter that may be modified by the called method. In addition, each formal parameter vertex is control dependent on the method entry vertex, and each actual parameter vertex is control dependent on the call vertex.

The construction of the standard SDG can be performed by connecting MDGs at call sites. A *call arc* is used to connect the call vertex and the entry 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 represent parameter passing. Moreover, to represent the *transitive flow of dependences* in the SDG, *summary arcs* [7] 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.

Existing SDGs for object-oriented programs can also represent object-oriented features such as classes and objects, class extensions, interfaces and their extensions, polymorphism, object parameters, and class libraries; details can be found in [11, 13, 14].

[**Example**] Figure 2 shows the SDG for the base code of the program in Figure 1, especially a part of class `Point`. In the figure, circles represent program statements and are labeled by statement numbers. Ellipses represent parameter vertices and are labeled with fi_in or fi_out for formal parameters and ai_in or ai_out for actual parameters. Following Larsen[13] we refer to a particular parameter vertex by prefixing the parameter label with the call or entry vertex upon which it is control dependent. For example, `s17->a3_in` refers to the parameter vertex representing the actual parameter "`_x_in=2`" in the call to `setX()` at `s17`. In the figure, we use different types of arcs to represent control dependences, data dependences, method calls, and parameter bindings. For example, there are control dependences arcs (`s17`, `a3_in`) and (`me9`, `f1_in`) since `a3_in` is an actual-in vertex attached to a method call to `setX()` at `s17` and `f1_in` is a formal-in vertex attached to the method entry vertex `me9`. `p` is defined in `s16` and used in `s17` and `s18`. Therefore, there are data dependence arcs (`s16`, `s17`) and (`s16`, `s18`). A parameter binding occurs at the call to `setY()` at `s18` between 2 in `main()` and `_y` in `setY()`. This parameter binding leads to a parameter-in arc (`s18->a4_in`, `m11->f4_in`) and a parameter-out arc (`m11->f2_out`, `s18->a6_out`). Finally,

[1] A directed graph $G = (V, A)$, where V is a set of vertices and $A \in V \times V$ is a set of arcs. Each arc $(v, v') \in A$ is directed from v to v'; we say that v is the source and v' the target of the arc.

there is a summary arc (`s16->a1_in`, `s16->a5_out`) to represent the fact that in `Point()` constructor, the value of 1 that is passed to `Point()` affects the value of `x` that is returned by `Point()`.

3. SYSTEM DEPENDENCE GRAPHS FOR ASPECT-ORIENTED PROGRAMS

We next present our system dependence graphs for aspect-oriented programs and our system dependence graph construction algorithm. We also discuss the representation of aspects, aspect and class interaction, and a complete aspect-oriented program using our representation.

3.1 Advice, Introduction, Pointcuts and Methods

In addition to methods, an aspect may contain other modular units such as advice and introduction. Since advice and introduction can be regarded as method-like units, we can also represent each piece of advice or introduction by a *method dependence graph* (MDG) as described in Section 2.2. To keep our terminology consistent in the rest of paper, we use the word "module" to stand for a piece of advice, a piece of introduction, or a method in an aspect and also a method in a class, and use a module dependence graph to represent it.

A *module dependence graph* (MDG) for a module m, denoted by G_{MDG}, is a directed graph (e, V, A) where e is an *entry vertex* to represent the entry into m; $V = V_n \cup V_c$ such that V_n is a set of *normal vertices* and V_c is a set of *call vertices*; $A = A_{con} \cup A_{dat}$ such that A_{con} is a set of *control dependence arcs* such that any $(u, v) \in A_{con}$ iff u is control-dependent on v ; A_{dat} is a set of *data dependence arcs* such that any $(u, v) \in A_{dat}$ iff u is data-dependent on v .

G_{MDG} is similar to the procedure dependence graph defined in [7] for representing a procedure in a procedural program. A vertex is called a *normal vertex* if it represents a statement or predicate expression in m without containing a call or object creation. Otherwise it is called a *call vertex*. G_{MDG} consists of two types of arcs, i.e., *control dependence arcs* and *data dependence arcs*. Let u and v be two statements in m, informally u is directly *control-dependent* on v if whether u is executed or not is directly determined by the evaluation result of v, and u is directly *data-dependent* on v if the value of a variable computed at v has a direct influence on the value of a variable computed at u.

For a piece of before, after, or around advice a, we use an MDG to represent a; the MDG has a unique entry vertex to represent the entry into a. To represent a piece of *introduction*, two cases should be considered. If a piece of introduction i introduces a method (or constructor) into a class, we use the MDG to represent i; the MDG has a unique entry vertex to represent the entry into i. If a piece of introduction i introduces a field f to a class, we need not construct the MDG for i. In such a case, we regarded f as an aspect instance variable (we discuss this issue in Section 3.2). For a pointcut pc, since it has no body code, we use a vertex called *join-point vertex* to represent pc; it also represents the entry into pc.

3.2 Base and Extended Aspects

We discuss how to use aspect dependence graphs to represent a base or extended aspect[2].

Base Aspects. To facilitate the analysis of an individual aspect, we represent each aspect in an aspect-oriented program by an aspect dependence graph.

Let α be an aspect with k modules $\{m_i | i = 1, 2, ..., k.\}$ and $G_i = (e_i, V_i, A_i)$ be the MDG for module m_i. An *aspect dependence graph* (ADG) for α, denoted by G_{ADG}, is a directed graph $(e^\alpha, \mathcal{E}^\alpha, \mathcal{V}^\alpha, \mathcal{A}^\alpha)$, where e^α is the *aspect entry vertex*; $\mathcal{E}^\alpha = \cup_{i=1}^{k} e_i$ is the set of *entry vertices* of the modules in α. $\mathcal{V}^\alpha = \cup_{i=1}^{k} V_i \cup V_{fi}^\alpha \cup V_{pc}^\alpha \cup V_{fo}^\alpha \cup V_{ai}^\alpha \cup V_{ao}^\alpha$ such that $\cup_{i=1}^{k} V_i$ is the set of vertices; each represents a statement or control predicate in the modules in α; V_{pc}^α is the set of *join-point vertices*; V_{fi}^α or V_{fo}^α is the set of *formal-in* or *formal-out vertices*; V_{ai}^α or V_{ao}^α is the set of *actual-in* or *actual-out vertices*. $\mathcal{A}^\alpha = \cup_{i=1}^{k} A_i \cup A_{ms}^\alpha \cup A_{pi}^\alpha \cup A_{po}^\alpha \cup A_c^\alpha cup A_p^p \cup A_s^\alpha$ such that $\cup_{i=1}^{k} A_i$ is the set of *control dependence arcs* and *data dependence arcs* in the MDGs of modules in α; $\mathcal{A}_{ms}^\alpha$ is the set of *membership arcs*; $\mathcal{A}_{pi}^\alpha$ or $\mathcal{A}_{po}^\alpha$ is the set of *parameter-in* or *parameter-out arcs*; $\mathcal{A}_c^\alpha$ is a set of *call arcs*; A_p^p is the set of *pointing arcs*; $\mathcal{A}_s^\alpha$ is a set of *summary arcs*.

G_{ADG} is a collection of MDGs; each represents a piece of advice, a piece of introduction, or a method in α. The *aspect entry vertex* represents the entry into α. An *aspect membership arc* represents the membership relationships between α and its members (advice, introduction, pointcuts, or methods) by connecting α's entry vertex to the entry vertex of each member. A *join-point vertex* represents a pointcut in α. *Formal-in* and *formal-out vertices* represent formal parameters in a module m; there is a formal-in vertex for each formal parameter of m and a formal-out vertex for each formal parameter that may be modified by m. Actual-in and actual-out vertices are used to represent actual parameters at each call site in a module m; there is an actual-in vertex for each actual parameter of m and an actual-out vertex for each actual parameter that may be modified by m. Each actual-in or actual-out vertex is control dependent on its corresponding call vertex and each formal-in or formal-out vertex is control dependent on the entry vertex.

A *call arc* represents the calling relationship[3] between two modules m_1 and m_2 in α by connecting the call vertex in m_1 to the entry vertex of m_2's MDG if there is a call in m_1's body to call m_2. A *parameter-in* or *parameter-out* arc represents the parameter passing between modules m_1 and m_2 in α by connecting an actual-in and a formal-in vertex to form a parameter-in arc, and a formal-out to an actual-out vertex to form a parameter-out arc if there is a call in m_1's body to call m_2. A *summary arc* models the transitive flow of dependence across a module call within α, as introduced in Section 2.2.

Consider an aspect instance variable *f* in α. Since the variable is accessible to all modules in α, we create a formal-in or formal-out vertex for each module in α that references *f*.

Finally, for each pointcut pc in α, we connect the aspect

[2]We can use an existing technique [13] to represent a base and extended class.

[3]Since advice in AspectJ is automatically woven into some method(s) by a compiler (called ajc) during aspect weaving process, there exists no call to the advice. As a result, there exists no call from introduction (or method) to advice.

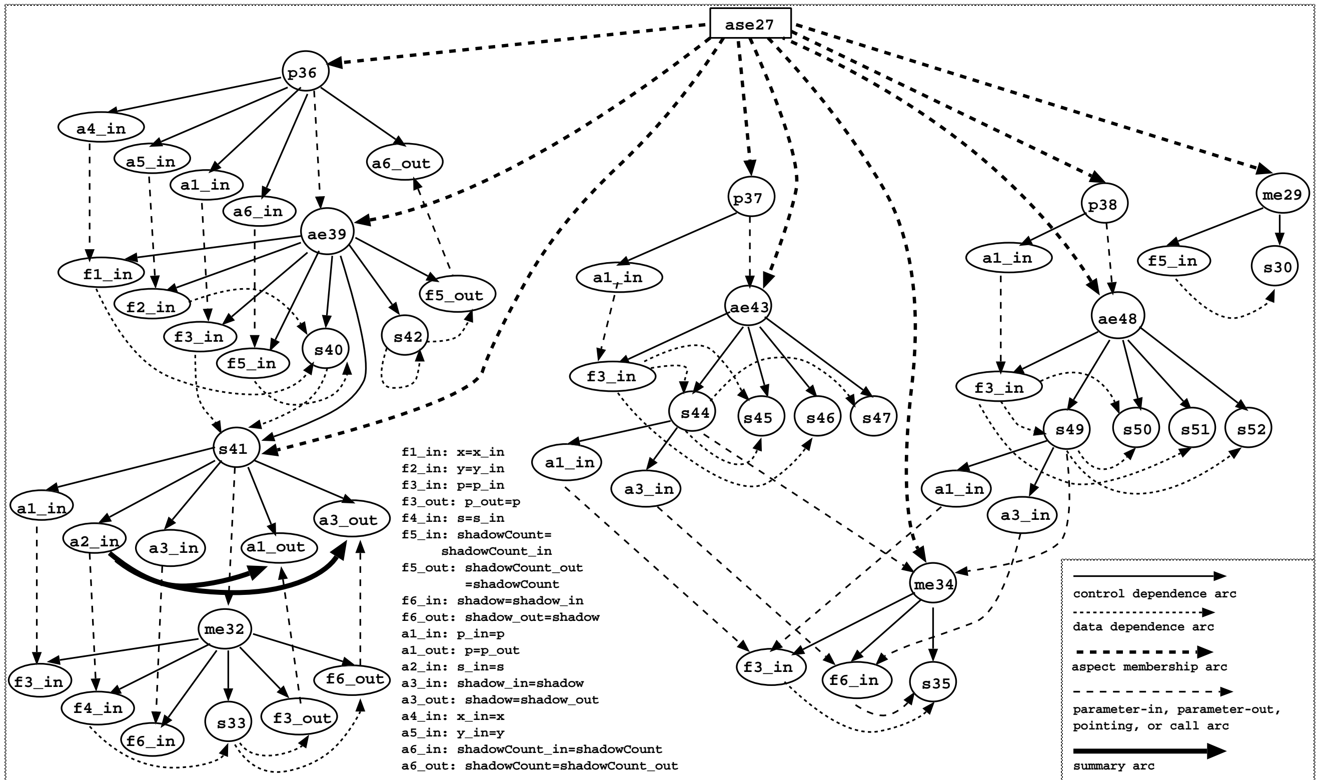

Figure 3: An ADG for aspect PointShadowProtocol in Figure 1.

entry vertex to *pc*'s join-point vertex through an aspect membership arc, and also *pc*'s join-point vertex to the entry vertex of its corresponding advice by a pointing arc to represent the relationship between them. Since there exists parameter passing between *pc* and its corresponding advice *a*, parameter-in and parameter-out arcs are created to connect an actual-in vertex to a formal-in vertex, and also a formal-out vertex to an actual-out vertex between *pc* and *a*.

[**Example**] Figure 3 shows the aspect dependence graph for aspect `PointShadow Protocol`. The rectangle represents the aspect entry vertex and is labeled by the statement label associated with the aspect entry. Circles represents statements and are labeled with the corresponding statement number in the aspect. For example, `ase27` is an aspect entry vertex; `ae39`, `ae43`, and `ae48` are advice entry vertices; `me29`, `me32`, and `me34` are method entry vertices, `p36`, `p37`, and `p38` are join-point vertices. (`ase27`, `ae39`), (`ase27`, `me32`), and (`ase27`, `pe36`) are aspect membership arcs. Each entry vertex is the root of a sub-graph which is itself a partial SDG. Each sub-graph may contain control and data dependences, call and parameter, pointing, and summary arcs. (`p36`,`ae39`), (`p37`,`ae43`), and (`p38`,`ae48`) are pointing arcs that represent interactions between pointcuts and their corresponding advice. To represent parameter passing between pointcut `p36` and its corresponding after-returning advice, formal-in and formal-out vertices are created for the formal parameters `x`, `y`, and `p` of the advice, and actual-in and actual-out vertices are created for the parameters `x`, `y`, and `p` of pointcut `setting`. Also parameter-in and parameter-out arcs are created to connect these formal and actual vertices.

Extended Aspects. An aspect can extend an abstract aspect[4], a class, and can implement any number of interfaces. The ADG should be able to represent an extended aspect. Given an abstract aspect α and an aspect α' that extends α, we can construct the ADG for α' as follows. We first construct the MDG for each module m in α', and then reuse the MDGs of all modules that are inherited from α. There is an entry vertex for α to represent the entry into α, and aspect-membership arcs are used to connect the aspect entry vertex to the entry vertex of each modules in α. The aspect-membership arcs are also used to connect the aspect entry vertex to the entry vertices of any module in α that are inherited by α'. We also use formal-in and formal-out vertices to model the parameter passing in α and instance variables in α and α' that may be referenced by a call to this module. Similarly, formal-out vertices for a module represent the module's formal parameters and instance variables in super or extended aspects that may be modified by a call to this module. Similarly, we can represent an aspect that extends a class or implements an interface using existing techniques originally developed for object-oriented programs [11, 13].

3.3 Interactions between Aspects and Classes

An aspect can interact with a class in several ways: by *object creation*, *method call*, *introduction declaration*, and *join point*. An SDG should be able to represent these interactions between aspects and classes.

Object Creation. A module m in an aspect α may create an object of a class C through a declaration or by using

[4]In AspectJ, an aspect can not extend a concrete aspect.

an operator such as `new`. At this time, there is an implicit call from m to C's constructor. To represent this implicit constructor call, a call arc is added to connect the call vertex in α at the site of object creation to the entry vertex e of the MDG of C's constructor. Also, at the call vertex, actual-in and actual-out vertices are added to match the formal-in and formal-out vertices in the MDG of C's constructor. For example, statement `s40` in Figure 1 represents an object creation of class `Shadow` in aspect `PointShadowProtocol`. To represent this object creation, in the SDG of Figure 4, a call vertex is created for `s40`; it is connected to the entry vertex `me22` of the `Point`'s constructor. Actual-in vertices (`a7_in` and `a8_in`) and actual-out vertices (`a10_out` and `a10_out`) for `s40` are also added to match the formal-in and formal-out vertices for `Point`'s constructor.

Method Call. An aspect α may have a call from its module m_1 to a method m_2 in the public interface of class C. In this case, a call arc is added to connect the call vertex of m_1 to the entry vertex of m_2's MDG, and parameter-in and parameter-out arcs are added to connect actual-in vertices to formal-in vertices, and formal-out vertices to actual-out vertices between m_1 and m_2. For example, statement `s45` in Figure 1 represents a call to method `getX()` of class `Point` in aspect `PointShadowProtocol`. To represent this method call, in the SDG of Figure 4, a call vertex is created for `s45`; it is connected to the entry vertex `me5` of method `setX()`. An actual-in vertex (`a7_in`) for `s45` is added to match the formal-in vertex for method `setX`.

Introduction Declaration. An aspect α may declare a piece of introduction i that publicly introduces one method (or constructor) into a class C. To represent such an interaction, the entry vertex of C's class dependence graph is connected to the entry vertex of i's MDG by a class membership arc because i can potentially affect the type structure of C. If a piece of introduction in α publicly introduces a field f into a class C, we regard f as an instance variable of both α and C. Therefore, f is accessible to all modules in α and C. In this case, we create a formal-in and formal-out vertex for f for each module in α and C in which f is referenced. However, if a piece of introduction in α privately introduces a field f into a class C, f is only accessible to all modules in α. In this case, we create a formal-in and formal-out vertex for f only for each module in α. For example, statement `s31` in Figure 1 represents a piece of introduction that privately introduce a field `shadow` into class `Point`; this means only code defined in `PointShadowProtocol` can access `shadow`. To represent this introduced field, in the SDG of Figure 4, only a formal-in vertex (`f9_in`) for `shadow` is created for method `getShadow()` that references `shadow`, and both a formal-in vertex (`f9_in`) and a formal-out vertex (`f9_out`) for `shadow` are created for method `associate()` that modifies `shadow`.

Join Point. An aspect may be woven into one or more classes at some join points, declared within *pointcuts* which are used in the definition of *advice*. By carefully examining the pointcuts and their associated advice, one can determine those methods that a piece of advice may advise. This information can be used to connect the base program and aspects. Just as an aspect weaves itself into the base program at some join points, we weave the MDGs for advice into the partial SDG at join-point vertices (Section 3.5 will discuss this issue in detail). For example, the after advice declared in aspect `PointShadowProtocol` (lines ae43–s47) of Figure 1 may weave into method `setX()` of class `Point`. To represent this weaving issue, in the SDG of Figure 4, a weaving arc (`me9`, `p37`) is created to connect the entry vertex `me9` for method `setX()` to the join-point vertex `p37` for pointcut `settingX`.

3.4 Complete Aspect-Oriented Programs

We next present a system dependence graph for a complete aspect-oriented program.

Let $\mathcal{P}$ be an aspect-oriented program with n modules $\{m_i | i = 1, 2, ..., n.\}$ and $G_i = (e_i, V_i, A_i)$ be the MDG for module m_i. An *system dependence graph* (SDG) for $\mathcal{P}$, denoted by G_{SDG}, is a directed graph $(\mathcal{E}^p, \mathcal{V}^p, \mathcal{A}^p)$, where $\mathcal{E}^p = \cup_{i=1}^{n} e_i$ is the set of *entry vertices* of the modules in $\mathcal{P}$. $\mathcal{V}^p = \cup_{i=1}^{n} V_i \cup V_{pc}^p \cup V_{fi}^p \cup V_{fo}^p \cup V_{ai}^p \cup V_{ao}^p$ such that $\cup_{i=1}^{k} V_i$ is the set of vertices; each represents a statement or control predicate in the modules in $\mathcal{P}$; V_{pc}^p is the set of *join-point vertices*; V_{fi}^p or V_{fo}^p is the set of *formal-in* or *formal-out vertices*; V_{ai}^p or V_{ao}^p is the set of *actual-in* or *actual-out vertices*. $\mathcal{A}^p = \cup_{i=1}^{k} A_i \cup A_{pi}^p \cup A_{po}^p \cup A_c^p \cup A_p^p \cup A_w^p \cup A_s^p$ such that $\cup_{i=1}^{k} A_i$ is the set of *control* and *data dependence arcs* in the MDGs of modules in P^p; A_{pi}^p or A_{po}^p is the set of *parameter-in* or *parameter-out arcs*; A_c^p is a set of *call arcs*; A_p^p is the set of *pointing arcs*; A_w^p is the set of *weaving arcs*; A_s^p is a set of *summary arcs*.

G_{SDG} is a collection of MDGs; each represents a `main()` method, a method of a class, a piece of advice or introduction, or a method of an aspect. G_{SDG} also contains some additional arcs to represent direct or indirect dependences between a call and the called module and transitive interprocedural data dependences.

G_{SDG} uses *join-point vertices* to represent pointcuts in $\mathcal{P}$. G_{SDG} also uses *formal-in* and *formal-out vertices* to represent formal parameters in a module, and *actual-in* and *actual-out vertices* to represent actual parameters at each call site. In G_{SDG}, *call arcs* represent the calling and callee relationships between modules. *Weaving arcs* connect the MDG for a method to the MDG for its corresponding advice; these arcs represent the relationships between advice and those methods that the advice may affect. 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* to model the parameter passing between modules. G_{SDG} uses *summary arcs* to represent the *transitive flow of dependences.*

[**Example**] Figure 4 shows the complete SDG for program in Figure 1, which was constructed by the algorithm described in Figure 5.

3.5 The Algorithm for SDG Construction

Our SDG construction algorithm for an aspect-oriented program $\mathcal{P}$ consists of four steps: (1) preprocessing each aspect and class; (2) building MDGs for modules; (3) connecting MDGs at callsites; (4) weaving MDGs at join points.

Figure 5 shows our SDG construction algorithm `BuildSDG`. As input `BuildSDG` gets the control flow graph for each modules in all aspects and classes of $\mathcal{P}$, and as output `BuildSDG` returns the $\mathcal{P}$'s SDG. First, `BuildSDG` preprocesses each as-

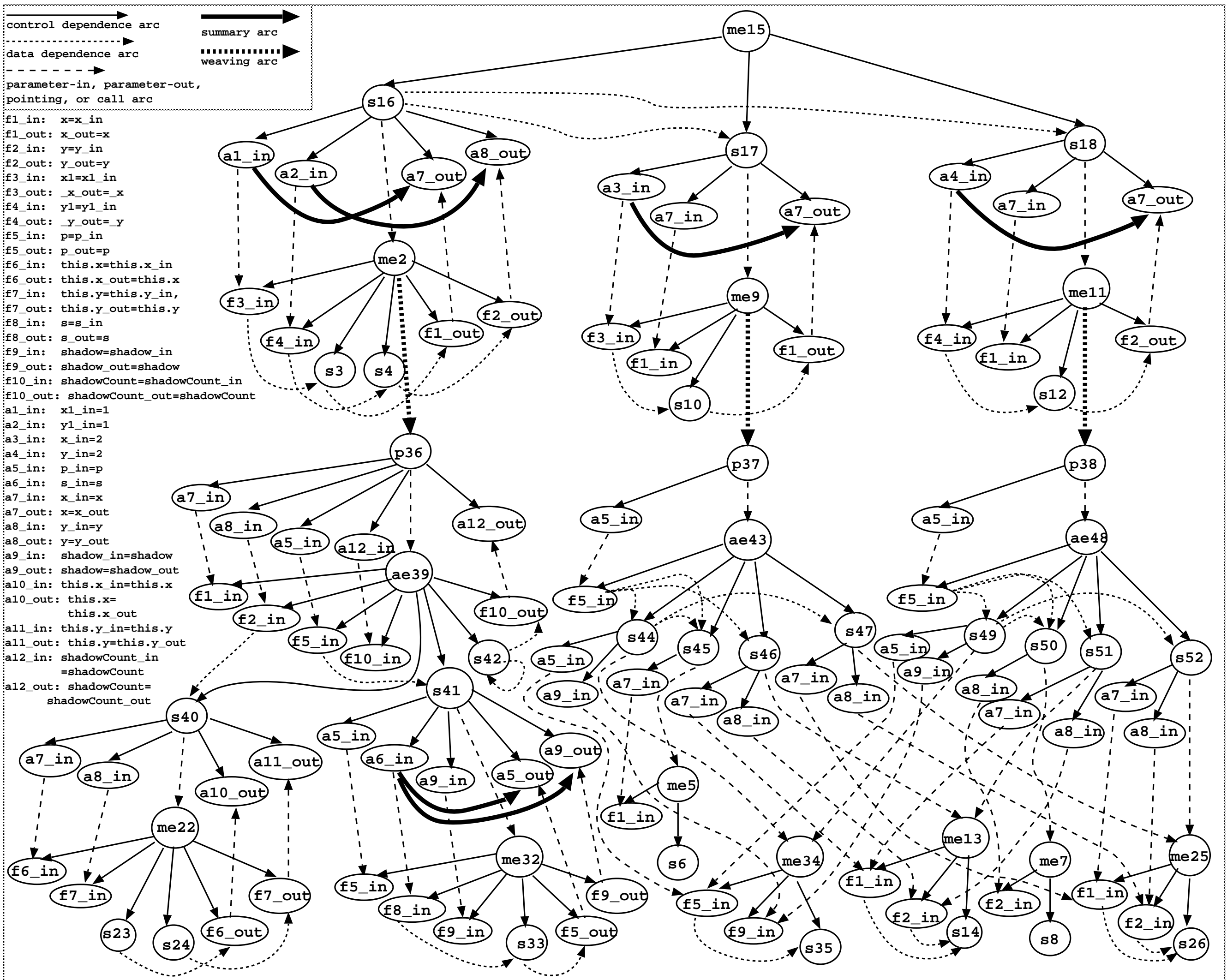

Figure 4: An SDG of the program in Figure 1.

pect and class in $\mathcal{P}$ to get those kinds of information that are necessary for constructing the SDG (lines 1-12). Second, `BuildSDG` builds the MDG for each piece of advice, introduction, and method in an aspect or class. It builds these MDGs in a bottom-up fashion according to the aspect and class hierarchies (lines 13-31). After that, `BuildSDG` calls `Connect()` to connect these MDGs at call sites to form a partial SDG for $\mathcal{P}$ (line 32). Finally, `BuildSDG` builds the complete SDG for $\mathcal{P}$ by (1) calling `Weaving()` to weave the MDG for each piece of advice into the MDGs for its corresponding methods in the partial SDG, and (2) calling `AddSummaryArcs()` to compute summary arcs at call sites regarding both aspects and classes (lines 33-34); we can use an existing technique presented in [18] to implement the procedure `AddSummaryArcs()`. In the following, we describe our algorithm step by step.

Step1: Preprocessing Aspects and Classes. In this step, `BuildSDG` first identifies advice, introduction, and methods that require new MDGs, and then computes the GREF and GMOD sets for each module. Finally, `BuildSDG` determines the *affected-method set* for each pointcut.

(1) *Identifying Advice, Introduction, and Methods Requiring New MDGs.* `BuildSDG` uses an existing algorithm [14] to identify methods in each class that require a new MDG. It builds on this idea to identify advice, introduction, and methods in each aspect that requires a new MDG. We describe this identification process only for aspects here; for classes, one can refer to [14].

For an aspect α, `BuildSDG` calls a marking procedure to operate on α's call graph to identify the advice, introduction, and methods that require new MDGs. First, it marks the advice, introduction, and methods declared in α. Second, if α extends some base aspects[5], it marks the advice, introduction, and methods in the base aspects that can reach these marked advice, introduction, and methods by performing a backward traversal on α's call graph from these marked advice, introduction, and methods. Finally, all marked advice,

[5]We can use a similar technique to handle the case that α is extended from a class or interface.

```
algorithm BuildSDG

input Control flow graphs (CFGs) of program P
output System dependence graph (SDG) of P
declare

begin BuildSDG
     /* Step 1: Preprocessing the program P */
  [ 1] foreach class C
  [ 2]    Identify the methods that need new MDGs
  [ 3] endfor
  [ 4] foreach aspect A
  [ 5]    Identify the advice, introduction, and methods that
           need new MDGs
  [ 6] endfor
  [ 7] foreach module m
  [ 8]    Compute GREF and GMOD sets for m
  [ 9] endfor
  [10] foreach pointcut pc
  [11]    Compute affected-method set for pc
  [12] endfor
     /* Step 2: Build MDGs for methods in each class and
        advice, introduction, and methods in each aspect */
  [13] foreach class C
  [14]    Construct an MDGs for each method in C
  [15] endfor
  [16] foreach aspect A
  [17]    foreach advice, introduction, or method m declared in A
  [18]       Compute control dependences for m
  [19]       Compute data dependences for m
  [20]       Expend objects in m
  [21]       Compute data dependences for objects
  [22]    endfor
  [23]    foreach advice, introduction, or method m
                 in the base aspects
  [24]       if m is "marked" then
  [25]          Copy old module dependence graph
  [26]          Adjust callsites
  [27]       else
  [28]          Reuse m's old module dependence graph
  [29]       endif
  [30]    endfor
  [31] endfor
     /* Step 3: Connecting MDGs at call sites */
  [32] Connect()
     /* Step 4: Weaving MDGs at pointcut sites */
  [33] Weaving()
  [34] AddSummaryArcs()
end BuildSDG
```

Figure 5: Algorithm for SDG construction.

introduction, and methods require new MDGs.

(2) *Determine GREF and GMOD Sets for Each Module.* BuildSDG uses α's call graph to compute the interface (i.e., two sets *GREF* and *GMOD*), for the advice, introduction, and methods that require new MDGs. For a piece of advice, introduction, or method m, *GREF(m)* is the set of non-local variables (i.e., parameters, instance variable, or data members) to which m refers, and *GMOD(m)* is the set of non-local variables that m might modify. Using a technique that is similar to the construction of a callsite or entry site for a method, which is described in Section 2.2, (1) a complete call site (i.e., call and actual parameter vertices) for a method call statement can be created, and (2) a complete entry site (i.e., entry and formal parameter vertices) for a piece of advice, introduction, or method m can be created based on the *GREF* and *GMOD* sets of m. *GREF(m)* and *GMOD(m)* can be determined by an existing data-flow analysis algorithm [12].

(3) *Determine Affected-Method Sets for Each Pointcut.* In this phase, BuildSDG calls PointcutAnalysis() to perform static analysis on each pointcut to determine the methods that the pointcut may affect. As input PointcutAnalysis() gets a pointcut pc, and as output PointcutAnalysis() returns a set called *affected-methods-set* which records methods that may be affected by pc.

Step 2: Building MDGs for Modules. In this step, BuildSDG uses an existing algorithm [14] to construct an MDG for each method in each class and uses a slightly modified version of the algorithm to construct an MDG for each piece of advice, introduction, and method in each aspect. We briefly describe the algorithm here; details can be found in [14].

BuildSDG constructs the module dependence graph for a piece of advice, introduction, or method m declared in a new aspect based on m's control flow graph. BuildSDG builds the control dependence graph for m using an existing algorithm [6], and builds the data dependence graph for m in three phases. First, BuildSDG preprocesses m using an algorithm that is a modified version of the usual data dependence graph construction algorithm [1]. Second, BuildSDG expands a vertex that references an object into the representations. Third, BuildSDG computes data dependences for data members of the objects. These three steps can handle objects at call sites, objects as parameters, and polymorphic objects. In addition, BuildSDG constructs the module dependence graph for a piece of advice, introduction, or method m declared in a base aspect using a similar way as described in [14].

Step 3: Connecting MDGs at Call Sites. In this step, BuildSDG connects the MDGs created in Step 2 at call sites to form a partial SDG for an aspect-oriented program. At each call site, BuildSDG connects the MDG for the called introduction or method to the MDG for the calling advice, introduction, or method using three types of arcs: (1) a call arc connects a call vertex to the entry vertex of a called introduction or method; (2) a parameter-in arc connects each actual-in vertex at the call site to the corresponding formal-in vertex; (3) a parameter-out arc connects each formal-out vertex to the corresponding actual-out vertex at the call site.

At each pointcut site, BuildSDG connects the join-point vertex for a pointcut to the entry vertex of its corresponding advice using three types of arcs: (1) a pointing arc connects a join-point vertex to the entry vertex of its corresponding advice; (2) a parameter-in arc connects each actual-in vertex at the pointcut site to the corresponding formal-in vertex at advice site; (3) a parameter-out arc connects each formal-out vertex at the advice site to the corresponding actual-out vertex at pointcut site. Moreover, if multiple pieces of advice apply to the same pointcut, BuildSDG connects the join-point vertex of the pointcut to the entry vertex of each piece of advice by pointing arcs respectively. In this case, we ignore the advice precedence because the connection is not related to the resolution order of the advice.

Step 4: Weaving MDGs at Join Points. To form the complete SDG, we need to know some join points in the MDGs for methods at which the MDGs for methods and their corresponding advice can be woven in a natural way. As we discussed previously, since an aspect-oriented program is woven at some join points specified in pointcuts, we can use join-point vertices for weaving MDGs for advice and

Table 1: Parameters contributing to the SDG size.

Parameters	Meanings of Parameters
γ_{vertex}	Largest number of statements in a single advice, introduction, or method
γ_{arc}	Largest number of arcs in a single advice, introduction, or method
γ_{param}	Largest number of formal parameters for any advice, introduction, or method
γ_{istant}	Largest number of instance variables in an aspect or class
$\gamma_{callsite}$	Largest number of call sites in any advice, introduction, or method
γ_{tree}	Depth of inheritance tree determining number of possible indirect call destinations
γ_{module}	Number of advice, introduction, and methods

methods. The task for weaving the complete SDG consists of two phases: (1) weave the MDGs for advice in aspects into the MDGs for their corresponding methods in classes; (2) add summary arcs to form the final complete SDG.

`Weaving()` connects the entry vertex of each method's MDG in the partial SDG to the join-point vertex of a pointcut that refers to the method by a *weaving arc*. If the advice attached to the pointcut is a piece of *around* advice that contains a `proceed()` clause, `Weaving()` connects the `proceed` call vertex to the entry vertex of the original method's MDG by a call arc to represent that the around advice may execute the `proceed()` call, which leads to execute the original method under the join point declared by the pointcut.

To model parameter passing between the method and its corresponding advice, `Weaving()` uses a similar technique as in phase (1). For each join-point vertex, actual-in and actual-out vertices are created for each parameter of the pointcut. For each entry vertex of the advice's MDG, formal-in and formal-out vertices are created for each formal parameter in advice. `Weaving()` uses parameter-in arcs to connect the actual-in vertices to the formal-in vertices, and uses parameter-out arcs to connect formal-out vertices to the actual-out vertices between the pointcut and the advice. `Weaving()` does this iteratively until all pieces of advice in all aspects have been processed.

3.6 The Size of SDG

The size of SDG is critical for applying it to the practical development environment for aspect-oriented programs. Here we try to predict the size of SDG based on the work of Larsen and Harrold [13], who gave an estimate for the size of the SDG for object-oriented programs.

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, advice, and introduction declarations. As a result, we can apply Larsen and Harrold's approximation here to estimate the size of the SDG of an aspect-oriented program by regarding an aspect as a class-like unit and a piece of advice or introduction as a method-like unit. However, since our SDG is constructed for an aspect-oriented program, its size may be different than the SDG of an object-oriented program.

Table 1 lists the parameters that contribute to the size of an SDG, denoted by G_{SDG}, for aspect-oriented programs. We give a bound on the number of parameters for any advice, introduction, or method m, denoted by $\Upsilon_{Param}(m)$, and use this bound to compute an upper bound on the size of a single advice, introduction, or method m, denoted by $\Upsilon_{Size}(m)$. Based on $\Upsilon_{Size}(m)$ and γ_{module} (number of advice, introduction, and methods in an aspect-oriented program), we can compute the upper bound on the number of vertices in G_{SDG} including all aspects and classes, denoted by $\Upsilon_{Size}(G_{SDG})$ that contribute to the size of the SDG of the program.

$$\Upsilon_{Param}(m) = \gamma_{param} + \gamma_{instant};$$
$$\Upsilon_{Size}(m) = O(\gamma_{vertex} + \gamma_{callsite} * (1 + \gamma_{tree} * (2 * \Upsilon_{Param}(m))) + 2 * \Upsilon_{Param}(m));$$
$$\Upsilon_{Size}(G_{SDG}) = O(\Upsilon_{Size}(m) * \gamma_{module}).$$

From the above estimate, we can see that $\Upsilon_{Size}(G_{SDG})$ provides only a rough upper bound on the number of vertices in an SDG. In practice an SDG may be considerably more space efficient.

4. RELATED WORK

We discuss related work in the area of developing dependence based representations (DBRs), a general class of program representations that includes SDGs.

DBRs have been primarily studied for procedural programs to support compiler optimization and develop software engineering tools. Ferrante *et al.* [6] use the *program dependence graph* (PDG) to explicitly represent control and data dependences in a procedural program. The envisioned uses are compiler optimizations and program slicing [20]. Horwitz *et al.* [7] extend the PDG to the *system dependence graph* (Horwitz-Reps-Binkley SDG for short) to represent a procedural program with multiple procedures. Cheng [5] presents a *process dependence net* which is the generalization of the PDG to represent program dependences in a concurrent procedural program. The goal is to support the development of concurrent programs. Krinke [10] uses a *threaded program dependence graph* that can present interference dependences in a concurrent program; the intended application is program slicing. Although these DBRs can effectively represent the features in procedural programs, they were not designed to represent object-oriented and aspect-oriented constructs.

DBRs have been applied to represent object-oriented programs recently. Larsen and Harrold [13] present a new system dependence graph (Larsen-Harrold SDG), that extends the Horwitz-Reps-Binkley SDG, to represent object-oriented programs for program slicing. Their SDG can represent many object-oriented features such as classes and objects, inheritance, polymorphism, and dynamic binding. Liang and Harrold [14] further extend the Larsen-Harrold SDG to represent object parameters, polymorphic objects, and class libraries. Kovacs *et al.* [11] extends the Larsen-Harrold SDG to represent Java programs. In addition to general object-oriented features, their SDG can represent some Java-specific features such as static variables, interfaces, and multiple packages. On the other hand, McGregor *et al.* [15] take a different way to develop a DBR called *object-oriented program dependency graph* (OPDG) for object-oriented programs. The OPDG is an extension of the PDG [6] and can be used as a basis for computing layered static slices for object-oriented programs. Chen *et al.* [4] also extends the PDG [6] to a *object-oriented dependency graph*

for representing object-oriented programs. These DBRs can effectively represent object-oriented programs, but are not designed to represent specific aspect-oriented features such as join points, advice, introduction, aspects, aspects, and aspect inheritance in aspect-oriented programs.

Recently, Zhao [21] presents a DBR called an *aspect-oriented system dependence graph* (ASDG) for slicing aspect-oriented programs. The ASDG consists of an SDG for the non-aspect code, a group of dependence graphs for the aspect code, and some additional vertices and arcs to represent dependences between aspects and classes. However, Zhao dose not give a concrete algorithm for constructing the ASDG, therefore how to represent aspect weaving in an SDG is not clear. We see our work differing from Zhao's in several ways. First, we give a concrete algorithm for constructing an SDG for aspect-oriented programs. Our algorithm gives an efficient way to weave dependence graphs for base and aspect code and to model the parameter passing between pointcuts and their corresponding advice. Second, in contrast to Zhao's approach (which requires the creation of some extra weaving vertices to facility the weaving of dependence graphs for base and aspect code), we use pointcuts as join points directly, then use pointing and weaving arcs to represent aspect weaving. This approach reflects the actions of an aspect weaver, which weaves the aspect code into the base code at join points. Third, our SDG can represent many more aspect-oriented constructs. These constructs include around advice, advice precedence, introduction scope, and aspect inheritance.

5. CONCLUDING REMARKS

In this paper we extend previous *system dependence graphs* (SDGs) to represent aspect-oriented programs and present an algorithm for constructing SDGs. Our SDG construction algorithm consists of several steps. It first constructs a *module dependence graph* (MDG) for each piece of advice, introduction, and method in each aspect and class. It then uses existing techniques for object-oriented programs to connect these MDGs at call sites to form a partial SDG. Finally, our algorithm weaves the MDG for each piece of advice into the partial SDG for those methods whose behavior may be affected by the advice. The result is the complete SDG. Our SDGs capture the additional structure present in many aspect-oriented features such as join points, advice, introduction, aspects, and aspect inheritance, and various types of interactions between aspects and classes. They also correctly reflect the semantics of aspect-oriented concepts such as advice precedence, introduction scope, and aspect weaving. SDGs therefore provide a solid foundation for the further analysis of aspect-oriented programs.

Acknowledgments We would like to thank Wes Isberg for valuable comments on an earlier version of the paper.

6. 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. 2002. `http://aspectj.org`

[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. 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.

[5] 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.

[6] 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.

[7] 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.

[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] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin, "An Overview of AspectJ," *proc. 13th European Conference on Object-Oriented Programming*, pp220-242, LNCS, Vol.1241, Springer-Verlag, June 2000.

[10] 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.

[11] G. Kovacs, F. Magyar, and T. Gyimothy, "Static Slicing of Java Programs," Technical Report TR-96-108, Research Group on Artificial Intelligence, Hungarian Academy of Sciences, December 1996.

[12] M. Landi, B. G. Ryder, and S. Zhang, "Interprocedural Modification Side Effect Analysis with Pointer Aliasing," *Proc. 1993 SIGPLAN Conference on Programming Language Design and Implementation*, pp.56-67, June 1993.

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

[14] D. Liang and M. J. Harrold, "Slicing Objects Using System Dependence Graph," *Proceeding of the International Conference on Software Maintenance*, November 1998.

[15] J. D. McGregor, B. A. Malloy, and R. L. Siegmund, "A Comprehensive Program Representation of Object-Oriented Software," *Annal of Software Engineering*, Vol.2, pp.51-91, 1996, Baltzer Science Publishers, 1996.

[16] 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.

[17] A. Podgurski and L. A. Clarke, "A Formal Model of Program Dependences and Its Implications for Software Testing, Debugging, and Maintenance," *IEEE Transaction on Software Engineering*, Vol.16, No.9, pp.965-979, 1990.

[18] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay, "Speeding Up slicing," *Proceeding of Second ACM Conference on Foundations of Software Engineering*, pp.11-20, December 1994.

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

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

[21] J. Zhao, "Slicing Aspect-Oriented Software," *Proc. 10th IEEE International Workshop on Program Comprehension*, pp.251-260, June 2002.