Control-Flow Analysis and Representation for Aspect-Oriented Programs

Jianjun Zhao
School of Software and Department of Computer Science and Engineering
Shanghai Jiao Tong University
800 Dongchuan Road, Shanghai 200240, China
zhao-jj@cs.sjtu.edu.cn

Abstract

Aspect-oriented programming (AOP) has been proposed as a technique for improving the separation of concerns in software design and implementation. The field of AOP has, so far, focused primarily on problem analysis, language design, and implementation. Even though the importance of program comprehension and software maintenance is known, it has received little attention in the aspect-oriented paradigm. However, as the software systems coded in AOP languages are accumulated, the development of techniques and tools to support program comprehension and software maintenance tasks for aspect-oriented software will become important. In order to understand and maintain aspect-oriented programs, abstract models for representing these programs are needed. In this paper, we present techniques to construct control-flow representations for aspect-oriented programs, and discuss some applications of the representations in a program comprehension and maintenance environment.

1 Introduction

Aspect-oriented programming (AOP) has been proposed as a technique for improving the separation of concerns in software design and implementation [3, 7, 9, 17]. AOP provides explicit mechanisms for capturing the structure of crosscutting aspects of the computation such as exception handling, synchronization, performance optimizations, and resource sharing. Because these aspects crosscut the dominant problem decomposition, they are usually difficult to express cleanly using standard languages and structuring techniques. AOP can eliminate the code tangling often associated with the use of such standard languages and techniques, making the program easier to develop, maintain, and evolve.

The field of AOP has, so far, focused primarily on problem analysis, language design, and implementation. Even though the importance of program comprehension and software maintenance is known, it has received little attention in the aspect-oriented paradigm. However, as the concept of the aspect-oriented programming (AOP) becomes acceptable, and the software systems coded in AOP languages are accumulated, the development of techniques and tools to support maintenance tasks such as program understanding and testing of aspect-oriented software will become important.

We believe that many program comprehension and software maintenance techniques that have been well developed for procedural and object-oriented programming languages are applicable, with more or less modification, to AOP languages. For example, many program comprehension and software maintenance techniques for procedural or object-oriented programming languages rely on control-flow information, which is typically gathered by the control-flow analysis and represented by a *call graph* or *control-flow graph* (CFG) [4]. The call graph and CFG, although originally proposed for compiler optimizations, have been used for various software engineering tasks such as program understanding, debugging, testing, maintenance, and complexity measurements [10, 11].

Control-flow analysis was originally considered for procedural programming languages to support compiler optimization and program comprehension and software maintenance, and it has been recently considered for object-oriented programming languages to accommodate specific object-oriented features such as classes and objects, inheritance, polymorphism, and dynamic binding [5, 12, 13, 15].

AOP languages present unique opportunities and problems for control-flow analysis and representation schemes because of its difference from procedural or object-oriented language. Although many control-flow analysis techniques have been proposed for procedural and object-oriented programs, little control-flow analysis technique has been proposed for aspect-oriented programs until now. Sereni and de Moor [14] consider constructing a call graph for a simple aspect-oriented programming language in order to perform static analysis of aspects. However, their technique does not

```
 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):
          target(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 = 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 1: An AspectJ program.

consider the construction of the control-flow graph which provide more detailed control-flow information than a call graph. Due to some specific features in aspect-oriented programs, existing control-flow analysis techniques and representations for procedural and object-oriented programs can not accommodate these features for an aspect-oriented program. In order to support program comprehension and software maintenance tasks such as regression test selection and software measurement for aspect-oriented programs, in this paper we present techniques to construct control-flow representations for aspect-oriented programs. The proposed control-flow representations can be used as an underlying basis to support program comprehension and software maintenance tasks such as regression test selection [13] and structural metrics [10, 11], and also some other analysis tasks like data-flow analysis and control dependence analysis.

Moreover, in considering control-flow analysis of aspect-oriented programs, our concern is to characterize only the aspect related parts of an aspect-oriented program. As a result, in this paper, we do not consider issues relating to those kinds of classes whose behavior may not be affected by some aspect in the program. The control-flow analysis of those language elements can be handled, for example, by using techniques proposed by Sinha and Harrold [15, 16].

The rest of the paper is organized as follows. Section 2 briefly introduces the aspect-oriented programming with AspectJ. Section 3 shows a motivational example. Section 4 considers the control-flow analysis for a partial aspect-oriented program, and presents the control-flow graph for an individual aspect. Section 5 considers the control-flow analysis for a complete aspect-oriented program, and presents the system control-flow graph. Section 6 discusses some applications of our control-flow representations. Concluding remarks are given in Section 7.

2 Aspect-Oriented Programming in AspectJ

We present our system control-flow graph in the context of AspectJ, the most widely used aspect-oriented programming language [8]. Our basic techniques, however, deal with the basic concepts of aspect-oriented programming and therefore apply to the general class of AOP languages.

AspectJ [8] 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, inter-type declaration, and aspect. We briefly introduce each of these constructs as follows.

The *aspect* is the modular unit of crosscutting implementation. Each aspect encapsulates functionality that crosscuts other classes in a program. An aspect can be instantiated, can contain states and methods, and also may be specialized with sub-aspects. An aspect is combined with the classes it crosscuts according to specifications given within the aspect. Moreover, an aspect can use an *inter-type declaration* construct to declare fields, methods, and interface implementation declarations for classes. Declared members may be made visible to all classes and aspects (public inter-type declaration) or only within the aspect (private inter-type declaration), 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`.

An aspect can also be declared as abstract, which means it can not be instantiated. By default, a concrete aspect has only one instance that exists for the program execution [2]. Also named pointcuts can be declared abstract within an abstract aspect, allowing them to be given concrete definitions within concrete sub-aspects, much as abstract methods are used.

A central concept in the composition of an aspect with other classes is called a *join point*. A join point is a well-defined point in the execution of a program, such as a call to a method, an access to an attribute, an object initialization, an exception handler, etc. Sets of join points may be represented by *pointcuts*, implying that such sets may crosscut the system. Pointcuts can be composed and new pointcut designators can be defined according to these combinations. AspectJ provides various pointcut *designators* that may be combined through logical operators to build up complete descriptions of pointcuts of interest. For example, the aspect `PointShadowProtocol` in Figure 1 declares three pointcuts named `setting`, `settingX`, and `settingY` at `p36`, `p37`, and `p38`. An aspect can specify *advice*, which is used to define code that executes when a pointcut is reached. Advice is a method-like mechanism which consists of instructions that execute *before*, *after*, or *around* a pointcut. `around` advice executes *in place* of the indicated pointcut, allowing a method to be replaced. For example, the aspect `PointShadowProtocol` in Figure 1 declares three pieces of after advice at `ae39`, `ae43`, and `ae48`; each is attached to the corresponding pointcut `setting`, `settingX`, or `settingY`.

An AspectJ program can be divided into two parts: *base 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*, which 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 an existing technique proposed by Harrold and Rothermel [6]. To simplify the presentation we present only those join points related to method and constructor calls.

3 Motivational Example

This section presents a motivation example to discuss the issues that arise in control-flow analysis and representation of aspect-oriented programs.

Consider the aspect `PointShadowProtocol` shown in Figure 1, `PointShadowProtocol`, that declares a piece of *after-advice* (with the `settingX` pointcut), or code to be executed before traversing a join point into a method body. This after-advice is applicable to each join point where a target object of type `Point` receives a call to the method with signature `Point.setX(int)`. The `target` keyword is used to give the name to the target object.

In AspectJ this after-advice is applied by the compiler without explicit reference to the aspect from the `Point` class. So when performing control-flow analysis on class `Point`, by definition, existing control-flow analysis approaches only consider the `Point` class itself, but does not consider the effect from the `PointShadowProtocol` aspect. However, when class `Point` and aspect `PointShadowProtocol` are compiled together, then intuitively the behavior of `Point`'s `setX` method may be changed due to the after-advice, that means, from the outside of the method, the control flow within method `setX` should include the part of the after-advice. As a result, in order to correctly perform the control-flow analysis of class `Point`, we should take into account not only the `Point` class itself but also those pieces of after-advice declared in `PointShadowProtocol` aspect, that may affect the control-flow behavior of `Point` through the after-advice. On the other hand, consider that we want to perform control-flow analysis on aspect `PointShadowProcotol`, we should not just focus on the aspect itself, because the after-advice in the aspect only specify a partial behavior of the methods declared in class `Point`, and also because the after-advice is automatically woven into some methods in the `Point` class by the compiler, and therefore no call exists for the after-advice. So when we perform control-flow analysis

on PointShadowProcotol, we should consider the PointShadowProcotol together with those methods in class Point, whose behavior may be affected by the advice from PointShadowProcotol. Unfortunately, existing control-flow analysis and representation techniques for procedural or object-oriented programming languages consider neither of these cases. Therefore, it is impractical to perform control-flow analysis on an aspect or class in isolation in an aspect-oriented program. Rather, we should perform control-flow analysis on an aspect together with some methods that are advised by the aspect's advice. Also we should perform control-flow analysis on a class together with advice that may affect the behavior of methods in the class. Furthermore, to perform the whole-program control-flow analysis on a complete aspect-oriented program, we should consider both of these cases to make all pieces of advice tailored to their corresponding methods through the whole analysis process.

Note that in this paper, we only consider join points related to method and constructor calls (we treat a constructor of a class as a special case of a method), and such inter-type declarations that introduce members such as methods and constructors. Also, we just consider before and after advice. We leave around dvice, other types of join points, and other inter-type declarations that introduces members such as fields as our future work.

4 Control-Flow Analysis and Representation for Partial Programs

We next discuss the issues related to partial-program control-flow analysis, i.e., control-flow analysis of an individual aspect of an aspect-oriented program. We also show how to construct the control-flow graph for an aspect.

4.1 Representing Individual Modules

In addition to methods, an aspect may contain other modular units such as advice and inter-type members. Since advice and inter-type members can be regarded as method-like units, to keep our terminology consistent in the rest of paper, we use the word "module" to stand for a piece of advice, an inter-type member, or a method in an aspect and also a method in a class.

A *control-flow graph* (CFG) for a module m, denoted by G_m, 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 is a set of *control flow arcs* to represent the flow of control between two vertices.

In G_m, 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_m can be used to represent the control flow information for a module of an aspect-oriented programs.

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* [2]. Since a piece of advice a can be regarded as a method-like unit, we can use a CFG to represent a. In this case, the CFG for a has a unique entry vertex to represent the entry into a.

Aspects can declare members (fields, methods, and constructors) that are owned by other types. These are called *inter-type* members. Aspects can also declare that other types implement new interfaces or extend a new class [2]. Since each of these inter-type members (only for a method or constructor) is similar in nature to a standard method or constructor, we can use a CFG to represent each of them. In this case, the CFG for an inter-type member has a unique entry vertex to represent the entry into the member.

For a pointcut pc, since it contains no body code, it does not need a CFG to represent it. In this case, we use a vertex called *join-point vertex* to represent pc. The join-point vertex also represents the entry into pc. As we will discuss in the following, a join-point vertex can be regarded as a "join point" to aid for weaving the CFGs for advice into the partial SCFG for base code.

4.2 Representing Base Aspects

To facilitate the analysis of an individual aspect, we represent each aspect in an aspect-oriented program by an *aspect control-flow graph*. The aspect control-flow graph represents the static control-flow relationships that exist within and among advice, inter-type members, and methods of an aspect.

Let α be an aspect with k modules $\{m_i \mid i = 1, 2, ..., k.\}$ and $G_i = (e_i, V_i, A_i)$ be the CFG for module m_i. An *aspect control-flow graph* (ACFG) for α, denoted by G_α, is a directed graph $(e^\alpha, \mathcal{E}^\alpha, \mathcal{V}^\alpha, \mathcal{A}^\alpha)$, where e^α is the *aspect entry vertex* and $\mathcal{E}^\alpha = \cup_{i=1}^{k} e_i$ is the set of *entry vertices* of the modules in α. $\mathcal{V}^\alpha = \cup_{i=1}^{k} V_i \cup V_{jp}^\alpha$ 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 and V_{jp}^α is the set of *join-point vertices*. $\mathcal{A}^\alpha = \cup_{i=1}^{k} A_i \cup A_{ms}^\alpha \cup A_c^\alpha \cup A_p^\alpha \cup A_s^\alpha$ such that $\cup_{i=1}^{k} A_i$ is the set of *control flow arcs* in the CFGs of modules in α, $\mathcal{A}_{ms}^\alpha$ is the set of *membership arcs*, $\mathcal{A}_c^\alpha$ is a set of *call arcs*, A_p^α is the set of *pointing arcs*, and $\mathcal{A}_w^\alpha$ is the set of *weaving arcs*.

G_α is a collection of CFGs; each represents a piece of advice, an inter-type member, 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, inter-type members, 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. A *call arc* represents the calling relationship[1] between

[1] 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

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 CFG if there is a call in m_1's body to call m_2. *Weaving arcs* represent advice weaving by connecting the CFG for a method in some classes to the CFG for its corresponding advice in α.

For each pointcut pc in α, we connect the aspect 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.

4.3 Representing Extended Aspects

An aspect can extend an abstract aspect[2], a class, and can also implement any number of interfaces. The ACFG should be able to represent an extended aspect. Given an abstract aspect α and an aspect α' that extends α, we can construct the ACFG for α' as follows. We first construct the CFG for each module m in α', and then reuse the CFGs 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'. Similarly, we can represent an aspect that extends a class or implements an interface.

Example 1 *Figure 2 shows the ACFG for aspect* `PointShadowProtocol`. *For example,* `ase27` *is an aspect entry vertex;* `ae39`, `ae43`, *and* `ae48` *are advice entry vertices;* `me29`, `me32`, *and* `me34` *are method entry vertices,* `pe36`, `pe37`, *and* `pe38` *are join-point vertices. (*`ase27`, `me29`*), (*`ase27`, `me32`*), and (*`ase27`, `me34`*) are aspect membership arcs. Each entry vertex is the root of a sub-graph which is itself a partial SCFG. Each sub-graph is a CFG that represents the control-flow information in a module. (*`p36`,`ae39`*), (*`p37`,`ae43`*), and (*`p38`,`ae48`*) are pointing arcs that represent interactions between pointcuts and their corresponding advice.*

5 Control-Flow Analysis and Representation for Complete Programs

We next discuss the issues related to the whole program control-flow analysis (i.e., for both non-aspect and aspect code) of a complete aspect-oriented program. We also show how to construct the whole interprocedural control-flow graph called *system control-flow graph* for an aspect-oriented program.

a result, there exists no call from an inter-type member (or method) to advice.

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

5.1 Representing Interactions between Aspects and Classes

In AspectJ, an aspect can interact with a class in several ways, i.e., by *object creation*, *method call*, and *advice weaving*. The system control-flow graph for an aspect-oriented program should be able to represent these interactions between aspects and classes.

Method Calls and Object Creations. In AspectJ, a call may occur between two modules m_1 and m_2 that can be a piece of advice, an inter-type member, or a method of aspects and classes. In such a case, a call arc is added to connect the call vertex of m_1's CFG to the entry vertex of m_2's CFG. On the other hand, a piece of advice, an inter-type member, or a method m in an aspect or a class α may create an object of a class C through a declaration or by using 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 CFG of C's constructor.

Example 2 *In Figure 1, statement* `s45` *represents a call to method* `getX()` *of class* `Point` *in aspect* `PointShadowProtocol`. *To represent this method call, in the SCFG of Figure 3, a call vertex is created for* `s45`*; it is connected to the entry vertex* `me5` *of method* `setX()` *by a call arc.*

Advice Weaving. In aspect-oriented language such as AspectJ, the join point model is a key element for providing the frame of reference that makes it possible for execution of a program's aspect and non-aspect code to be coordinated properly. We recognized that the join point model is also a crucial point to perform interprocedural control-flow analysis for aspect-oriented programs because control-flow analysis of aspect and non-aspect code of the program is not independent. Rather, they must be coordinated through the join points (declared by *pointcut* designators) in the program. As a result, properly handling join points in the aspect code is a key for performing interprocedural control-flow analysis of an aspect-oriented program.

To form the complete SCFG, we need to know some "join points" in the CFGs for some methods at which the CFGs for their corresponding advice can be woven. By performing a static analysis for a pointcut declaration, we can determine those methods in some classes that a piece of advice, attached to this pointcut, may advise. This information can be used to connect the partial SCFG for base code to the CFGs for the aspect code; just as an aspect weaves itself into the base program at some join points, we weave the CFGs for advice into the partial SCFG at join-point vertices.

The basic idea of our approach is that we treat a piece of advice as a method-like unit when constructing the SCFG

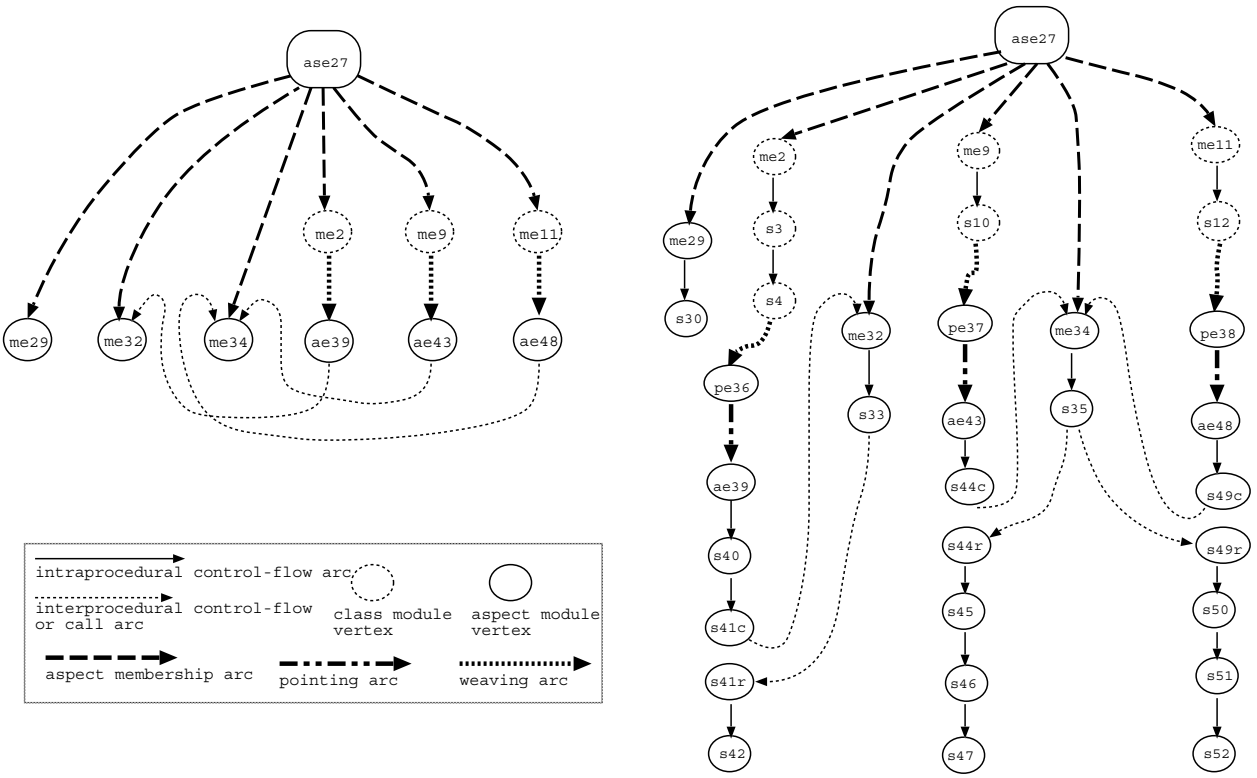

Figure 2: The ACG and ACFG corresponding to aspect `PointShadowProtocol`.

for an aspect-oriented program and regard each pointcut as a join point for weaving the CFGs of advice and the partial SCFG for base code. For a piece of before or after advice a in an aspect that may advise a method m in a class, we connect the entry vertex of m (advised method) to the join point vertex attached by a using a *weaving arc*. This is similar to the case that m contains a method call, i.e., we treat a together with its pointcut(s) as a method that may be called from m. The weaving arc here is similar to a call arc, but with different meaning.

Based on these considerations, we can weave the CFGs for advice and the partial SCFG to form the complete SCFG in a natural way.

Example 3 *The after advice (lines* `ae43-s47`*) in aspect* `PointShadowProtocol` *may weave into method* `setX()` *of class* `Point`*. To represent this weaving issue, in the SCFG of Figure 3, a weaving arc (*`s10`*,* `pe37`*) is created to connect statement* `s10` *of method* `setX()` *to the join-point vertex* `pe37` *for pointcut* `settingX`*.*

5.2 Representing a Complete Aspect-Oriented Program

We use the *system control-flow graph* to represent the control-flow information and calling relationships in a complete aspect-oriented program.

Let $\mathcal{P}$ be an aspect-oriented program with n modules $\{m_i \mid i = 1, 2, ..., n.\}$ and $G_i = (e_i, V_i, A_i)$ be the CFG for module m_i. A *system control-flow graph* (SCFG) for $\mathcal{P}$, denoted by $G_{\mathcal{P}}$, 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 $\mathcal{P}$. $\mathcal{V}^p = \cup_{i=1}^{n} V_i \cup V_{jp}^p$ such that $\cup_{i=1}^{k} V_i$ is the set of vertices; each represents a statement or control predicate in the modules in $\mathcal{P}$ and V_{jp}^p is the set of *join-point vertices*. $\mathcal{A}^p = \cup_{i=1}^{k} A_i \cup A_c^p \cup A_p^p \cup A_w^p$ such that $\cup_{i=1}^{k} A_i$ is the set of *control flow arcs* in the CFGs of modules in P^p, A_c^p is a set of *call arcs*, A_p^p is the set of *pointing arcs*, and A_w^p is the set of *weaving arcs*.

$G_{\mathcal{P}}$ is a collection of CFGs; each represents a `main()` method, a method of a class, a piece of advice, an inter-type member, or a method of an aspect. $G_{\mathcal{P}}$ also contains some additional arcs to represent calling relationships between a call and the called module and aspect weaving. $G_{\mathcal{P}}$ uses a *join-point vertex* to represent a pointcut in $\mathcal{P}$. In $G_{\mathcal{P}}$, *call arcs* represent the calling and callee relationships between modules. *Weaving arcs* connect the CFG for a method to the CFG for its corresponding advice; these arcs represent the weaving relationships between advice and those methods that the advice may affect.

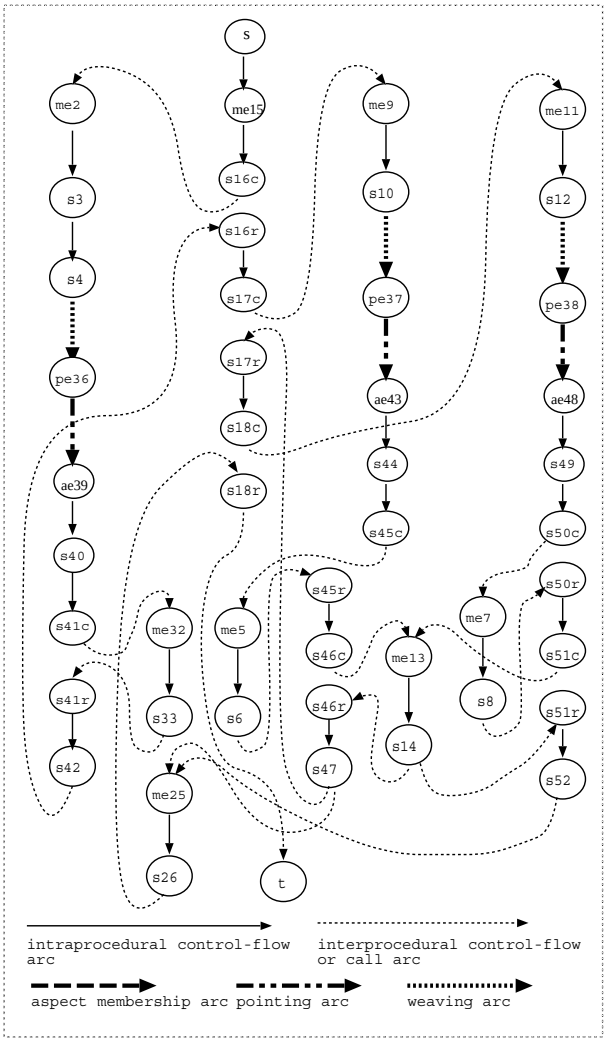

Figure 3: The complete SCFG for the program in Figure 1.

Example 4 *Figure 3 shows the SCFG for the program in Figure 1 with aspect* `PointShadowProtocol`.

6 Applications of CFGs

The CFGs presented in previous sections can be used in many software engineering tasks for aspect-oriented software. Here we briefly describe two tasks: structural testing and structural complexity measurement.

6.1 Structural Testing and Coverage

Structural testing techniques use a program's structure to guide the selection of test cases. Since our CFG represents execution paths in aspect-oriented programs, they can be used to define control-flow based coverage criteria, i.e., test data selection rules based on covering execution paths, for testing aspect-oriented programs [18, 19]. Therefore using our representations for structural testing provides an accurate measure of the adequacy of a test suite.

6.2 Structural Metrics

Software metrics are useful in software engineering tasks such as program understanding, testing, and project management. One could imagine that once some metrics could be proposed for aspect-oriented programs, they should be helpful in the development of aspect-oriented software. Because the CFG of an aspect-oriented program represents control-flow properties in the program, based on the CFG, we can define some structural metrics for measuring the complexity of the aspect-oriented program from different viewpoints. For example, we can extend the McCabe's control-flow based metrics [10] to the case of aspect-oriented software.

7 Concluding Remarks

In this paper we presented techniques to construct control-flow representations for aspect-oriented programs. The proposed representations can be used as a basis for supporting maintenance tasks such as regression test selection and structural metrics, and also some other analysis tasks like data-flow analysis and control dependence analysis.

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 control-flow analysis tool may differ from one language to another because each language has its own structure and syntax which must be handled carefully. As our future research, we would like to extend our control-flow analysis technique to cover around advice, other types of join points, and other inter-type declarations that introduces members such as fields. We are developing a control-flow analysis tool for AspectJ to automatically construct the ACFG and SCFG for aspect-oriented programs.

References

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

[2] The AspectJ Team, "The AspectJ Programming Guide," 2003. `http://aspectj.org`

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

[4] 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.

[5] D. Grove, G. DeFouw, J. Dean, and C. Chamber, "Call Graph Construction in Object-Oriented Languages," *Proceeding of OOPSLA'97*, 1997.

[6] M. J. Harrold and G. Rothermel, “A Coherent Family of Analyzable Graph Representations for Object-Oriented Software”, Technical Report OSU-CISRC-11/96-TR60, Department of Computer and Information Science, The Ohio State University, November 1996.

[7] 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.

[8] 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.

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

[10] J. McCabe, “A Complexity Measure,” *IEEE Transactions on Software Engineering*, Vol.2, No.4, pp.308-320, December 1976.

[11] 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.

[12] D. Rayside, S. Reuss, E. Hedges, and K. Kontogiannis, “The Effect of Call Graph Construction Algorithms for Object-Oriented Programs on Automatic Clustering,” *Proceeding of the International Conference on Software Maintenance*, pp.11-20, November 1998.

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

[14] D. Sereni and O. de Moor, “Static Analysis of Aspects,” *Proceeding of the International Conference on Aspect-Oriented Software Development* (AOSD’2003), March 2003.

[15] S. Sinha and M. J. Harrold, “Analysis of Programs with Exception-Handling Constructs,” *Proceeding of the International Conference on Software Maintenance*, pp.11-20, November 1998.

[16] S. Sinha and M. J. Harrold, “Analysis and Testing of Programs with Exception-Handling Constructs,” *IEEE Transactions on Software Engineering*, Vol.26, No.9, pp.849-871, September 2000.

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

[18] T. Xie and J. Zhao. A Framework and Tool Supports for Generating Test Inputs of AspectJ Programs. In Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD 2006), Bonn, Germany, pp. 190-201, March 2006.

[19] J. Zhao, “Data-Flow-Based Unit Testing of Aspect-Oriented Programs,” *Proc. 27th Annual IEEE International Computer Software and Applications Conference*, pp.188-197. Dallas, Texas, USA, November 2003.