2004－SE－145　(12)
2004／8／20

Towards Regression Test Selection for AspectJ Programs

Jianjun Zhao
Department of Computer Science and Engineering
Fukuoka Institute of Technology
3-30-1 Wajiro-Higashi, Higashi-ku, Fukuoka 811-0295, Japan
zhao@cs.fit.ac.jp

Abstract

The current research so far in aspect-oriented software development is focused on problem analysis, software design, and implementation techniques. Even though the importance of software testing is known, it has received little attention in the aspect-oriented paradigm. This paper presents the first safe regression test selection technique for AspectJ programs. The technique is based on various types of control flow graphs that can be used to select test cases, from the original test suite, that execute code that has been changed for the new version of the AspectJ software. The technique is code-based in the sense that it operates on the control flow graphs of AspectJ programs. The technique can be applied to modified individual aspects or classes, and also the whole programs that used modified aspects or classes.

1 Introduction

Aspect-oriented software development (AOSD) is a new technique to support separation of concerns in software development [4, 7, 11, 16]. The techniques of AOSD make it possible to modularize crosscutting aspects of a system. Like objects in object-oriented software development, aspects in AOSD may arise at any stage of the software life cycle, including requirements specification, design, implementation, etc. Some examples of crosscutting aspects are exception handling, synchronization, and resource sharing.

The current research so far in AOSD is focused on problem analysis, software design, and implementation techniques. Even though the importance of software testing and verification is known, it has received little attention in the aspect-oriented paradigm. Although it has been claimed that applying an AOSD method will eventually lead to quality software, aspect-orientation does not provide correctness by itself. An aspect-oriented design can lead to a better system architecture and an aspect-oriented programming language enforces a disciplined coding style, but they are by no means shields against programmer's mistakes or a lack of understanding of the specification. As a result, software testing remains an important task even in AOSD.

Regression testing is a necessary and important activity at both testing and maintenance phases. Regression testing aims at showing that code has not been adversely affected by modification activities during maintenance. Regression test selection techniques reuse tests from an existing test suite to test a modified program. By reusing such test suites to retest modified programs maintainers (testers) can reduce the effort which is required to perform that testing.

Aspect-oriented programming introduces some new language constructs such as join points, advice, introduction, aspects, that differ from procedural and object-oriented programs. These specific constructs in aspect-oriented programs require special testing support and provide opportunities for exploitation by a testing strategy. However, although many regression test selection techniques have been proposed for procedural programs [3, 5, 13] and object-oriented programs [9, 10, 14, 6], there is no regression test selection technique for aspect-oriented programs until now. Also, the existing regression test selection techniques can not be directly applied to aspect-oriented programs. Therefore, new regression test selection techniques and tools that are appropriate for aspect-oriented programs are needed.

This paper presents the first safe regression test selection technique for AspectJ programs. The technique is based on various types of control flow graphs that can be used to select test cases, from the original test suite, that execute code that has been changed for the new version of the AspectJ software. The technique is code-based in the sense that it operates on the control flow graphs of AspectJ programs. The technique can be applied to modified individual aspects or classes, and also the whole programs that used modified aspects or classes.

The rest of the paper is organized as follows. Section 2 briefly introduces the AspectJ. Section 3 presents a control flow model for regression test selection of AspectJ programs. 4 briefly describes a regression test selection algo-

rithm for AspectJ programs. Concluding remarks are given in Section 5.

2 Aspect-Oriented Programming in AspectJ

We present our data-flow-based unit testing approach of aspect-oriented programs 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 aspect-oriented 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 point, 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 sub-aspects. 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 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].

3 The Control Flow Model for AspectJ

To facilitate regression test selection for AspectJ programs, we present a control flow model that captures the control flow information of an aspect or class, and also a complete AspectJ program. Based on this model, regression test selection can be performed. The model consists of two different types of control flow graphs in order to capture different levels of control flow information in an individual aspect or class, and also the whole program. We present each type of the graphs as follows.

3.1 Modeling 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_{CFG}, 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 set of *control flow arcs* to represent the flow of control between two vertices.

In G_{CFG}, 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_{CFG} 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 before, after, or around 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.

```
 ce0 public class Point {
  s1   protected int x, y;
 me2   public Point(int _x, int _y) {
  s3     x = _x;
  s4     y = _y;
       }
 me5   public int getX() {
  s6     return x;
       }
 me7   public int getY() {
  s8     return y;
       }
 me9   public void setX(int _x) {
 s10     x = _x;
       }
me11   public void setY(int _y) {
 s12     y = _y;
       }
me13   public void printPosition() {
 s14     System.out.println("Point at("+x+","+y+")");
       }
me15   public static void main(String[] args) {
 s16     Point p = new Point(1,1);
 s17     p.setX(2);
 s18     p.setY(2);
       }
     }
ce19 class Shadow {
 s20   public static final int offset = 10;
 s21   public int x, y;

me22   Shadow(int x, int y) {
 s23     this.x = x;
 s24     this.y = y;
me25   public void printPosition() {
 s26     System.outprintln("Shadow at
               ("+x+","+y+")");
       }
     }
```

```
ase27 aspect PointShadowProtocol {
  s28   private int shadowCount = 0;
 me29   public static int getShadowCount() {
  s30     return PointShadowProtocol.
                 aspectOf().shadowCount;
       }
  s31   private Shadow Point.shadow;
 me32   public static void associate(Point p, Shadow s){
  s33     p.shadow = s;
       }
 me34   public static Shadow getShadow(Point p) {
  s35     return p.shadow;
       }

 pe36   pointcut setting(int x, int y, Point p):
          args(x,y) && call(Point.new(int,int));
 pe37   pointcut settingX(Point p):
          target(p) && call(void Point.setX(int));
 pe38   pointcut settingY(Point p):
          target(p) && call(void Point.setY(int));

 ae39   after(int x, int y, Point p) returning :
            setting(x, y, p) {
  s40     Shadow s = new Shadow(x,y);
  s41     associate(p,s);
  s42     shadowCount++;
       }
 ae43   after(Point p): settingX(p) {
  s44     Shadow s = new getShadow(p);
  s45     s.x = p.getX() + Shadow.offset;
  s46     p.printPosition();
  s47     s.printPosition();
       }
 ae48   after(Point p): settingY(p) {
  s49     Shadow s = new getShadow(p);
  s50     s.y = p.getY() + Shadow.offset;
  s51     p.printPosition();
  s52     s.printPosition();
     }
   }
```

Figure 1: A sample AspectJ program.

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 control flow graph 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.

3.2 Modeling Individual 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* (ACFG) 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_{ACFG}, 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*. $\mathcal{A}^\alpha = \cup_{i=1}^{k} A_i \cup A_{ms}^\alpha \cup A_c^\alpha \cup A_p^p \cup A_s^\alpha$ 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^p is the set of *pointing arcs*, and $\mathcal{A}_w^\alpha$ is the set of *weaving arcs*.

G_{ACFG} 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 two modules m_1 and m_2 in α by connecting

[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

```
algorithm BuildACFG
  input An aspect α
  output The Aspect Control-Flow Graph (ACFG) G_α of α
  begin BuildACFG
[ 1]  /* Build the aspect call graph for α and add to G_α */
[ 2]    G_α = Construct the aspect call graph for α
[ 3]  /* Build CFGs for advice, inter-type members,
[ 4]                    and methods in α and add to G_α */
[ 5]  /* Replace each call graph vertex with the corresponding CFG */
[ 6]    for each advice, inter-type member, or method m in α do
[ 7]      Replace α's aspect call graph vertex in G_α with m's CFG
[ 8]      Update arcs appropriately
[ 9]    endfor
[10]  /* Replace call sites with call and return vertices */
[11]    for each call vertex s in G_α, representing a call
[12]              to advice, inter-type member, or method m in α do
[13]      Replace s with a call and a return vertex
[14]      Update arcs appropriately
[15]    endfor
[16]  /* Connect the individual CFGs */
[17]    for each advice, inter-type member, or method m in α do
[18]      Add an arc from the aspect start vertex to the start
[19]      vertex of m's CFG in G_α
[20]    endfor
[21]  /* Return the complete ACFG of α */
[22]        return G_α
  end BuildACFG
```

Figure 2: Algorithm for ACFG construction.

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; we will discuss this issue in more detail in section 3.6.

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.

3.3 ACFG Construction Algorithms

Figure 2 gives an algorithm `BuildACFG` for building the ACFG for an aspect α which consists of five steps. As input `BuildACFG` gets CFGs for advice, inter-type members, and methods in α, and as output `BuildACFG` returns the α's ACFG.

First, `BuildACFG` builds the aspect call graph for α. An *aspect call graph* (ACG) for an aspect α represents caller/callee relationships among advice, inter-type members, and/or methods within α. Vertices in the ACG represent the advice, inter-type members, methods that are members of α, and those methods which are members of some classes and may be advised by advice of α. Arcs in the ACG represent the calling relationships among advice, inter-type members, and/or methods. ACG has a unique vertex called *aspect start vertex* to represent the entry into the aspect. ACG uses *aspect-membership arcs* to connect the aspect start vertex to each vertex that represents a piece of advice, an inter-type member, or a method in α. If an inter-type member or a method m_1 in α calls another inter-type member or method m_2 in α, ACG uses a *call arc* to connect m_1 to m_2 to represent the calling relationship between them. Second, `BuildACFG` builds CFGs for all advice, inter-type members, and methods using traditional algorithms like [1]. Third, `BuildACFG` replaces each vertex (except the start vertex) in the ACG with the corresponding CFG. Forth, `BuildACFG` replaces each call (site) vertex with a call and return vertices. Finally, `BuildACFG` connects the individual CFGs belonging to α to form the whole ACFG for α. If a module m_1 has a call to another module m_2 in α, `BuildACFG` connects the call vertex in m_1 to m_2's start vertex using a call arc.

Example 1 *Figure 3 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,* `p36`, `p37`, *and* `p38` *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.*

3.4 Modeling Aspect-Class Interactions

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* `s40` *represents an object creation of class* `Shadow` *in aspect* `PointShadowProtocol`. *To represent this object*

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

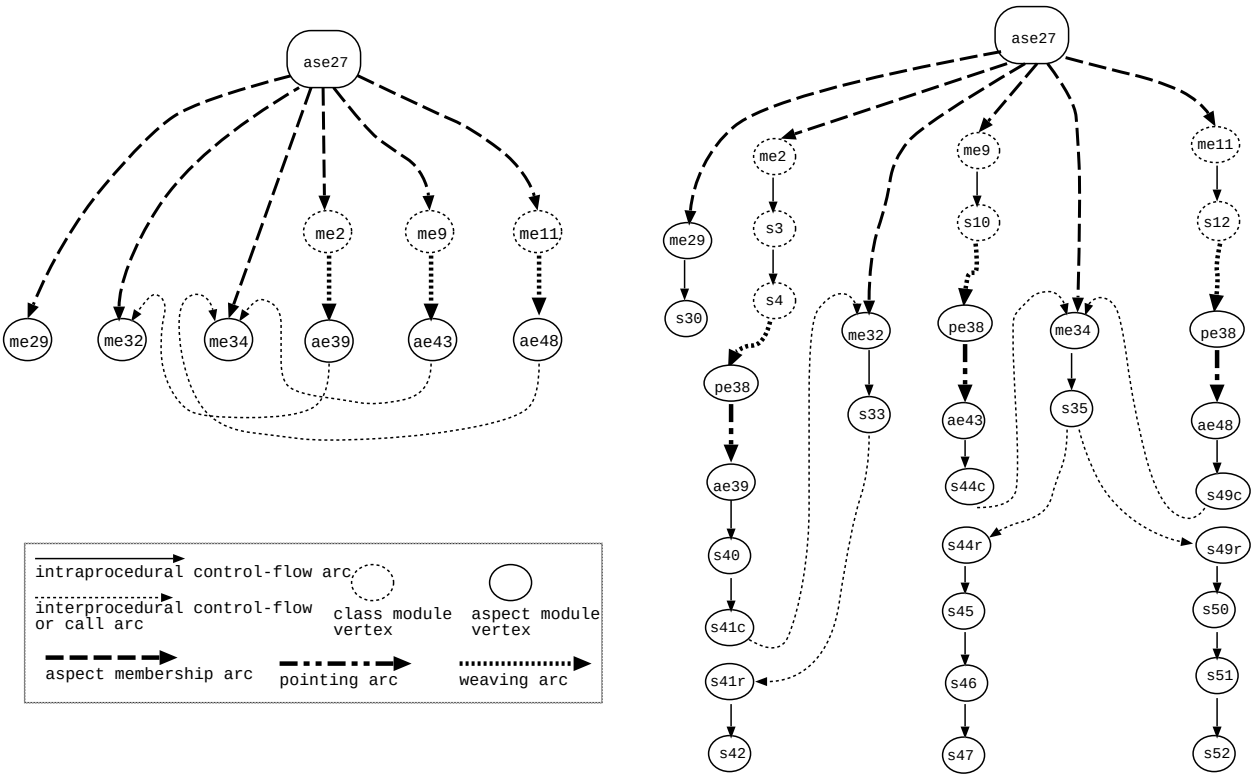

Figure 3: The ACG and ACFG corresponding to aspect `PointShadowProtocol`.

creation, in the SCFG of Figure 4, a call vertex is created for `s40`*; it is connected to the entry vertex* `me22` *of the* `Point`*'s constructor by a call arc. On the other hand, statement* `s45` *represents a call to method* `getX()` *of class* `Point` *in aspect* `PointShadowProtocol`*. To represent this method call, in the SCFG of Figure 4, 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 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. For a piece of around advice a in an aspect that may advise a method m in a class, since a may replace m, we add a weaving arc which connects the start vertex of the original call arc to m to the join-point vertex attached by a.

Based on these considerations, we can weave the CFGs

for advice and the partial SCFG to form the complete SCFG in a nature 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 4, a weaving arc (*`me9`, `pe37`*) is created to connect the entry vertex* `me9` *for method* `setX()` *to the join-point vertex* `pe37` *for pointcut* `settingX`*.*

3.5 Modeling Complete Programs

We use the *system control-flow graph* (SCFG) 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_{SCFG}, 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_{SCFG} 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_{SDG} also contains some additional arcs to represent calling relationships between a call and the called module and aspect weaving. G_{SCFG} uses a *join-point vertex* to represent a pointcut in $\mathcal{P}$. In G_{SCFG}, *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.

Example 4 *Figure 4 shows the SCFG for the program in Figure 1 with aspect* `PointShadowProtocol` *which can be constructed by the algorithm described in Figure 5.*

3.6 SCFG Construction Algorithm

We next present a concrete algorithm for constructing the system control-flow graph for an aspect-oriented program $\mathcal{P}$.

Figure 5 shows our SCFG construction algorithm `BuildSCFG`. As input `BuildSCFG` gets each module in all aspects and classes of $\mathcal{P}$, and as output `BuildSCFG` returns the $\mathcal{P}$'s SCFG. Our algorithm consists of four steps. First, `BuildSCFG` pre-processes each aspect and class in $\mathcal{P}$ to get those kinds of information that are necessary for constructing the SCFG (lines 1-7). Second, `BuildSCFG`

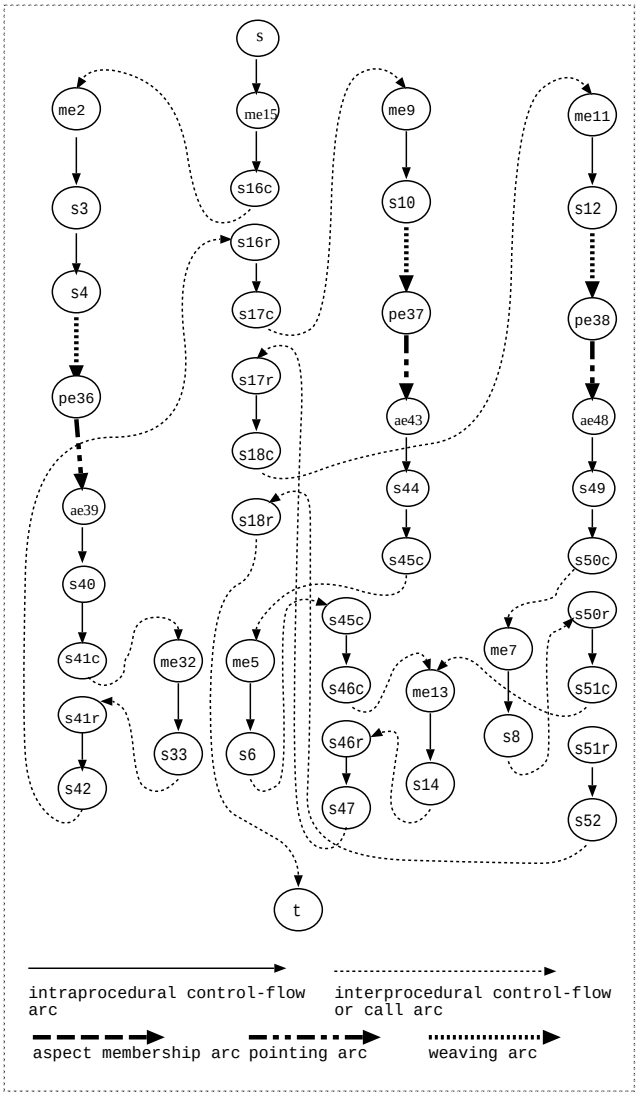

Figure 4: The complete SCFG for the program in Figure 1.

builds a CFG for each piece of advice, inter-type member, or method in an aspect or class. It builds these graphs in a bottom-up fashion according to the aspect and class hierarchies (lines 8-21). After that, `BuildSCFG` calls `Connect()` to connect these graphs at call sites to form a partial SCFG for $\mathcal{P}$ (line 22). Finally, `BuildSCFG` builds the complete SCFG for $\mathcal{P}$ by calling `Weaving()` to weave the CFG for each piece of advice into the CFGs for its corresponding methods in the partial SCFG (line 23). In the following, we describe our algorithm step by step.

Pre-processing Aspects and Classes. `BuildSCFG` first identifies pieces of advice, inter-type members, and methods that require new CFGs. `BuildSCFG` uses the following process to identify pieces of advice, inter-type members, and methods in each aspect that requires a new CFG; we can

```
algorithm BuildSCFG

input An aspect-oriented program P
output System control-flow graph (SCFG) of P
declare

begin BuildSCFG
/* Step 1: Pre-processing the program P */
[ 1] foreach aspect α or class C
[ 2]    Identify pieces of advice, inter-type members, and methods
[ 3]    that need new CFGs
[ 4] endfor
[ 5] foreach pointcut pc
[ 6]    Compute affected-methods set for pc
[ 7] endfor
/* Step 2: Build CFGs for pieces of advice, inter-type members
           and methods in each aspect or class */
[ 8] foreach aspect α or class C
[ 9]    foreach piece of advice, inter-type member, or method m
                declared in α or C
[10]       Compute the CFG for m
[11]    endfor
[12]    foreach piece of advice, inter-type member, or method m
[13]             in the base aspects or classes
[14]       if m is "marked" then
[15]          Copy old CFG
[16]          Adjust callsites
[17]       else
[18]          Reuse m's old CFG
[19]       endif
[20]    endfor
[21] endfor
/* Step 3: Connecting CFGs at call sites */
[22] Connect()
/* Step 4: Weaving CFGs at pointcut sites */
[23] Weaving()
end BuildSCFG
```

Figure 5: Algorithm for SCFG construction.

use a similar process to identify methods in each class that require a new CFG.

For an aspect α, `BuildSCFG` calls a marking procedure to operate on α's call graph to identify the pieces of advice, inter-type members, and methods that require new CFGs; the call graph for α can be constructed by a modified algorithm proposed in [15]. First, it marks the pieces of advice, inter-type members, and methods declared in α. Second, if α extends some base aspects[2], it marks the pieces of advice, inter-type members, and methods in the base aspects that can reach these marked advice, inter-type members, and methods by performing a backward traversal on α's call graph from these marked advice, inter-type members, and methods. Finally, all marked advice, inter-type members, and methods require new CFGs.

`BuildSCFG` then 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.

Building CFGs for Advice, Inter-type Members, and Methods. `BuildSCFG` uses an existing algorithm [1] to construct the CFG for a piece of advice, an inter-type member, or a method m declared in a new aspect or class and the CFG for a piece of advice, an inter-type member, or a method declared in a base aspect.

Connecting CFGs at Call Sites. `BuildSCFG` calls `Connect()` to connect the CFGs created in step 2 at call sites to form a partial SCFG for an aspect-oriented program. At each call site, `BuildSCFG` connects the CFG for the called inter-type member or method to the CFG for the calling advice, inter-type member, or method by using a call arc. At each pointcut site, `BuildSCFG` connects the join-point vertex for a pointcut to the entry vertex of its corresponding advice by using a pointing arc. If there are multiple pieces of advice that applies to the same pointcut, `BuildSCFG` connects the join-point vertex of the pointcut to the entry vertex of each piece of advice by pointing arcs respectively.

Weaving CFGs at Pointcut Sites. `BuildSCFG` calls `Weaving()` to finish the task of weaving the complete SCFG by weaving the CFGs for advice in aspects into the CFGs for their corresponding methods in classes. `Weaving()` connects the entry vertex of each method's CFG in the partial SCFG 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` call, `Weaving()` connects the `proceed` call vertex to the entry vertex of the original method's CFG 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. `Weaving()` does this iteratively until all pieces of advice in all aspects have been processed.

4 Regression Test Selection for AspectJ Programs

In order to perform the regression test selection for AspectJ programs, we adapt the graph-traversal algorithm proposed by Harrold et al. [6] and Rothermel *et al.* [14], which uses a control-flow-based representation of the original and modified versions of the software to select the test cases to be rerun. Our regression test selection for AspectJ programs takes the following steps:

- Run the test suites with the original program and obtain coverage information.
- Construct the system control-flow graph for the origi-

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

nal and modified programs.

- Compare the system control-flow graphs and detect dangerous arcs in the graphs.
- Compare the coverage information and dangerous arcs, and select test cases.

5 Concluding Remarks

This paper presented the first safe regression test selection technique for AspectJ programs. Our technique is based on various types of control flow graphs that can be used to select test cases, from the original test suite, that execute code that has been changed for the new version of the AspectJ software. Our technique is code-based in the sense that it operates on the control flow graphs of AspectJ programs. Our technique can be applied to modified individual aspects or classes, and also the whole programs that used modified aspects or classes. In our future work, we plan to develop a regression test selection tool based on the technique proposed in this paper to support regression test selection for AspectJ programs.

References

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

[2] The AspectJ Team. The AspectJ Programming Guide. August 2003.

[3] T. Ball. On the Limit of Control Flow Analysis for Regression Test Selection. *Proc. ACM International Symposium on Software Testing and Analysis*, pp.134-142, March 1998.

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

[5] Y. F. Chen, D. S. Rosenblum, and K. V. Vo. TestTube: A System for Selective Regression Testing. *Proc. 16th International Conference on Software Engineering*, pp.211-222, May 1994.

[6] M. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression Test Selection for Java Software. *Proc. ACM COnference on Object-Oriented Programming, Systems, Languages, and Applications*, pp.312-326, October 2001.

[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*, pp.220-242, LNCS, Vol.1241, Springer-Verlag, June 2000.

[9] T. Koju, S. Takada, N. Doi. Regression Test Selection based on Intermediate Code for Virtual Machines. *Proc. International Conference on Software Maintenance*, 2003.

[10] D. Kung, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen. Firewall Regression Testing and Software Maintenance. *Journal of Object-Oriented Programming*, 1994.

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

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

[13] G. Rothermel and M. J. Harrold. A Safe, Efficient Regression Test Selection Technique. *ACM Transactions on Software Engineering and Methodology*, Vol. 6, No. 2, pp.173-210, April 1997.

[14] G. Rothermel, M. J. Harrold, and J. Dedhia. Regression Test Selection for C++ Software. *Journal of Software Testing, Verification, and Reliability*, Vol. 10, No. 6, pp.77-109, June 2000.

[15] D. Sereni and O. de Moor. Static Analysis of Aspects. *Proc. 2nd International Conference on Aspect-Oriented Software Development*, pp.30-39, March 2003.

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

[17] L. J. White and K. Abdullah. A Firewall Approach for Regression Testing of Object-Oriented Software. *Proc. 10th Annual Software Quality Week*, May 1997.

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

[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.