Towards Regression Test Selection for AspectJ Programs

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

Tao Xie
Department of Computer Science
North Carolina State University
Raleigh, NC 27695
xie@csc.ncsu.edu

Nan Li
School of Software
Shanghai Jiao Tong University
800 Dongchuan Road,
Shanghai 200240, China
arctic@sjtu.edu.cn

ABSTRACT

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 a test suite to retest modified programs, maintainers or testers can reduce the required testing effort. This paper presents a regression test selection technique for AspectJ programs. The technique is based on various types of control flow graphs that can be used to select from the original test suite test cases that execute changed code for the new version of the AspectJ program. The code-base technique operates on the control flow graphs of AspectJ programs. The technique can be applied to modified individual aspects or classes as well as the whole program that uses modified aspects or classes.

1. INTRODUCTION

Aspect-oriented software development (AOSD) is a new approach to support separation of concerns in software development [5, 8, 12, 20]. The techniques in AOSD allow 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.

Most existing research in AOSD is focused on problem analysis, software design, and implementation techniques. Although software testing is important, it has received little attention in the aspect-oriented paradigm. Although it has been claimed that applying an AOSD technique 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 in AOSD.

Regression testing is a necessary and important activity at both testing and maintenance phases. It aims at showing that code has not been adversely affected by modification activities during maintenance. The selection techniques of regression test reuse tests from an existing test suite to test a modified program which can reduce the required testing effort of programs' maintainers or testers.

Aspect-oriented programming languages such as AspectJ introduce some new language constructs, such as join points, advice, introduction, and aspects, that differ from those in procedural and object-oriented programs. These specific constructs in aspect-oriented programs require special testing support and provide opportunities for being exploited by a testing strategy. However, although many regression test selection techniques have been proposed for procedural programs [4, 6, 14] and object-oriented programs [10, 11, 15, 7], there exists no regression test selection technique for aspect-oriented programs. In addition, the existing regression test selection techniques can not be directly applied to aspect-oriented programs, which have some unique features that must be considered specially during regression testing. Therefore, there is a need for new regression test selection techniques and tools that are appropriate for aspect-oriented programs.

This paper presents a regression test selection technique for AspectJ programs. The technique is based on various types of control flow graphs that can be used to select from the original test suite test cases that execute changed code for the new version of an AspectJ program. The code-based technique operates on the control flow graphs of AspectJ programs. The technique can be applied to modified individual aspects or classes as well as the whole program that uses modified aspects or classes.

The rest of the paper is organized as follows. Section 2 briefly introduces AspectJ. Section 3 briefly introduces the regression test selection techniques for Java programs. Section 4 presents our regression test selection technique for AspectJ programs. Section 5 briefly discusses related work and Section 6 concludes.

2. ASPECTJ

We present our regression test selection technique 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 aspect-oriented languages.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
WTAOP '06 Portland, Maine USA
Copyright 2006 ACM 1-59593-415-4/06/0007 ...$5.00.

```
 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.

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 point, pointcut, advice, introduction, and aspect. We briefly introduce each of these constructs as follows.

An *aspect* is a 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 that 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` shown 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, and an exception handler. 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 `pe36`, `pe37`, and `pe38`.

An aspect can specify *advice*, which is used to define code that executes when a pointcut is reached. Advice is a method-like mechanism that 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`, which are attached to the corresponding pointcut `setting`, `settingX`, and `settingY`, respectively.

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, containing the aspect `PointShadowProtocol`, which 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. A key related component is an *aspect weaver*, which ensures that applicable advice runs at appropriate join points. More information about AspectJ can be found elsewhere [3].

3. SAFE REGRESSION TEST SELECTION FOR JAVA PROGRAMS

We next briefly introduce the safe regression test selection technique developed by Harrold *et al.* [7] for Java programs; we develop our technique for AspectJ programs based on their technique. The key notion of their technique for java software to perform safe regression test selection are a control flow based representation called *Java Interclass Graph* (JIG) and a technique to detect dangerous arcs on the JIG. Basically, the technique takes the following steps:

- Run the test suites with the original program and obtain coverage information.

- Construct the Java Interclass Graph (JIG) for the original and modified programs.
- Compare the JIGs and detect dangerous arcs in the graphs.
- Compare the coverage information and dangerous arcs, and select test cases.

We next describe JIG and a technique to detect dangerous arcs on the JIG.

3.1 Java Interclass Graph

The *Java Interclass Graph* (JIG) [7] for a Java program is a collection of method control flow graphs each of which represents a `main()` method or a method in a class of the program. Additional arcs represent specific Java features, for example, method calls from the internal and external code, and exception handling. A *Control Flow Graph* (CFG) for a method m is a directed graph whose vertices represent statements or predicate expressions in m; arcs represent *control flow information.*

A representation of method calls from internal code is similar to most of traditional techniques such as those presented in [14]. To model interprocedural interactions between internal and external methods, these techniques use a special vertex called *external code vertex* to represent external code that may call an internal method or be called by an internal method. To do so, these techniques create arcs from the external code vertex to the entry vertices of some internal classes if these classes are accessible from external code. In addition, they create call arcs to connect the class entry vertex to the corresponding internal method because internal classes accessible from external code are considered to have internal methods that can be used to override external methods. These techniques also use a special vertex called *default vertex* to represent those methods that are not overridden by the internal methods. Special call arcs labeled with "*" are created from the class entry vertex to the default vertex.

To model exception handling in Java, these techniques use a similar way as the one proposed by Sinha and Harrold [16]. Their JIG can explicitly represents the *try*, *catch*, and *finally* blocks in each *try* statement.

3.2 Dangerous-Arc Detection

In the JIG of the original program, dangerous arcs are those arcs that represent control flows corresponding to the changed parts. Given the JIGs for the original and modified Java programs, Harrold *et al.* [7] use a depth-first algorithm, which is based on algorithms proposed by Rothermel *et al.* [14, 15], to detect dangerous arcs in the JIG of the original Java program by traversing in parallel the JIGs of the original and modified programs. The arc is regarded as a dangerous arc if the targets of the CFG arcs in the original and modified programs differ. Any test cases that cover the dangerous arcs must be rerun, and these test cases should be selected safely for regression testing because their behavior may be changed during maintenance and evolution phases. On the other hand, according to the algorithm developed by Harrold *et al.* [7], if an arc that is labeled as '*'becomes a dangerous arc, all test cases that create the instance of that corresponding class must be rerun as well.

4. SAFE REGRESSION TEST SELECTION FOR ASPECTJ PROGRAMS

We next present our safe regression test selection technique for AspectJ programs. Our technique is based on a similar technique developed by Harrold *et al.* [7] to select regression tests for Java programs (described in Section 3). The basic notion of the technique is to detect the dangerous arcs based on the control flow graphs of an original AspectJ programs and its modified version in order to safely select regression tests that must be rerun during the regression testing. However, existing control flow based representations such as JIG can not accommodate some specific features in AspectJ programs; therefore, we need to construct new control flow graphs for AspectJ. As a result, to facilitate selecting regression tests for AspectJ programs, we present a control flow model that captures the control flow information of an aspect and also a complete AspectJ program. Based on this model, we can perform regression test selection for AspectJ Programs.

4.1 Control Flow Model for AspectJ

Our control flow model consists of two different types of control flow graphs in order to capture different levels of control flow information in an individual aspect and the whole program. We present each type of the graphs as follows.

4.1.1 Modeling Individual Modules

In AspectJ, in addition to methods, an aspect may contain other modular units such as advice and inter-type members. Because advice and inter-type members can be treated as method-like units, to keep our terminology consistent in the rest of paper, we use the word "module" to refer to 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 is 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 AspectJ program.

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* [3]. Because a piece of before, after, or around advice a can be treated 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 [3]. Because 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.

Because a pointcut pc contains no body code, a control flow graph is not needed to represent pc. 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 is discussed later, a join-point vertex can be treated as a "join point" to help weave the CFGs for advice into the partial system control flow graph (described in Section 4.1.4 in details) for base code.

With the CFG as a representation of each individual module of an aspect, our technique can identify the changes in a piece of advice, method, or intertype method in the aspect.

4.1.2 Modeling Individual Aspects

To facilitate the analysis of an individual aspect, we represent each aspect in an AspectJ program with 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 α. An ACFG is a collection of CFGs, each of which 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 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 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.

With the ACFG as the representation of an individual aspect, our technique can identify the changes in the aspect by traversing the ACFGs of the original and modified aspects.

EXAMPLE 1. *Figure 2 shows the control flow graph 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 CFG that represents the control flow information in a module. (*`pe36`*,*`ae39`*), (*`pe37`*,*`ae43`*), and (*`pe38`*,*`ae48`*) are pointing arcs that represent interactions between pointcuts and their corresponding advice.*

4.1.3 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*. Our control flow model for an AspectJ program can 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, each of which 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 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, we add a call arc 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. This representation forms a partial CFG of an complete AspectJ program.

[1]Because advice in AspectJ is automatically woven into some method(s) by a compiler (such as ajc) during the aspect weaving process, there exists no call to the advice. As a result, there exists no call from an inter-type member (or method) to advice.

EXAMPLE 2. *In Figure 1, statement* `s40` *represents an object creation of class* `Shadow` *in the* `PointShadowProtocol` *aspect. To represent this object creation, in the SCFG of Figure 3, we create a call vertex for* `s40`*; the call vertex 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 3, we create a call vertex for* `s45`*; the call vertex is connected to the entry vertex* `me5` *of method* `setX()` *by a call arc.*

Advice Weaving. In AspectJ, the join point model is a key element for providing the frame of reference that coordinates properly the execution of a program's aspect and non-aspect code. We recognized that the join point model is also a crucial point to perform interprocedural control flow analysis for AspectJ programs because control flow analysis of aspect and non-aspect code of the program is not independent. Rather, aspect and non-aspect code 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 AspectJ program.

To build a complete control flow graph for n AspectJ program, we need to know some "join points" in the CFGs for some methods into which the CFGs for their corresponding advice can be woven. By performing a static analysis for a pointcut declaration, we can determine in some classes those methods that a piece of advice (being 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 notion of our approach is that we treat a piece of advice as a method-like unit when constructing the CFG for an AspectJ program and treat each pointcut as a join point for weaving the CFGs of advice and the partial CFG 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 weaving case 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 a different meaning. For a piece of around advice a in an aspect that may advise a method m in a class, because a may replace m, we add a weaving arc that connects the start vertex of the original call arc to m to the join-point vertex attached by a.

With this partial CFG generated, our technique can identify the weaving arcs (caused by advice weaving) that may be affected by a code change by traversing the partial CFGs

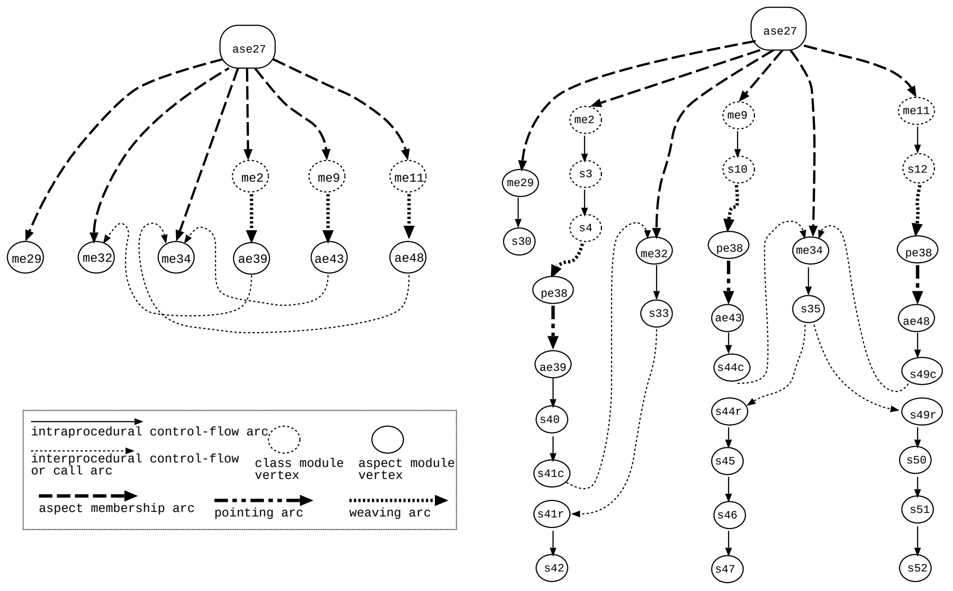

Figure 2: The ACG and ACFG corresponding to aspect `PointShadowProtocol`.

of the original and modified programs.

Example 3. *The after advice (lines* `ae43-s47`*) in aspect* `PointShadowProtocol` *may be woven into method* `setX()` *of class* `Point`*. To represent this weaving issue, in the SCFG of Figure 3, we create a weaving arc (*`me9`*,* `pe37`*) to connect the entry vertex* `me9` *for method* `setX()` *to the join-point vertex* `pe37` *for pointcut* `settingX`*.*

4.1.4 Modeling Complete Programs

We use the *system control flow graph* (SCFG) to represent the control flow information and calling relationships in a complete AspectJ program. An 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. The SCFG also contains some additional arcs to represent aspect weaving and calling relationships between a call and the called module. The SCFG uses a *join-point vertex* to represent a pointcut in $\mathcal{P}$. In the 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 3 shows the SCFG for the program in Figure 1 with aspect* `PointShadowProtocol`*.*

4.2 Detecting Dangerous Arcs

Having SCFGs for an original AspectJ program and its modified version, we can use the existing depth-first search algorithms developed in [7, 15] to detect the dangerous arcs of the original program by traversing in parallel the SCFGs of the original and modified programs. The arc is regarded as a dangerous arc if the targets of the SCFG arcs in the original and modified programs differ. Any test cases that cover the dangerous arcs must be rerun, and these test cases should be selected safely for regression testing of an AspectJ program because their behavior may be changed during maintenance and evolution phases.

5. RELATED WORK

We discuss related work in testing aspect-oriented programs and in developing safe regression test selection techniques based on control flow graphs.

Research on testing aspect-oriented programs has been mainly focused on code-based unit and integration testing, specification-based testing, automated test case generation, and fault model used for testing [2, 17, 18, 19, 21, 22]. Although regression testing is important, it is still received little attention in the aspect-oriented paradigm. To the best of our knowledge, our technique proposed in this paper is the first regression test selection technique for AspectJ programs.

Various approaches have been proposed to select regression tests for procedural and object-oriented programs [4, 6, 14, 10, 11, 15, 7]. Among these approach, Harrold *et al.* [7] propose a technique for safe regression test selection for Java programs. Their technique is based on a control flow based representation called *Java Interclass Graph* (JIG) to explicitly represent various specific features in Java programs. Based on the JIG, they use a depth-first search algorithm to detect dangerous edges for the ordinal programs and select test cases by comparing the coverage information. Koju *et al.* [10] propose a technique for regression test selection based on the Microsoft Intermediate Language (MSIL). Their technique is based on the one developed by Harrold *et al.* [7] for Java. They present control flow graphs to handle .Net-specific features such as delegate and present a class hierarchy analysis technique to support the regression test selection.

Although these safe regression test selection techniques can be used for Java programs and intermediate languages such as Microsoft Intermediate Language (MSIL), they can not be applied directly to AspectJ programs because of some specific features such as pointcut, advice, intertype declarations, and aspects. Our technique, which is based on the one developed by Harrold *et al.* [7] for Java, can handle regression-test selection problems that are unique to As-

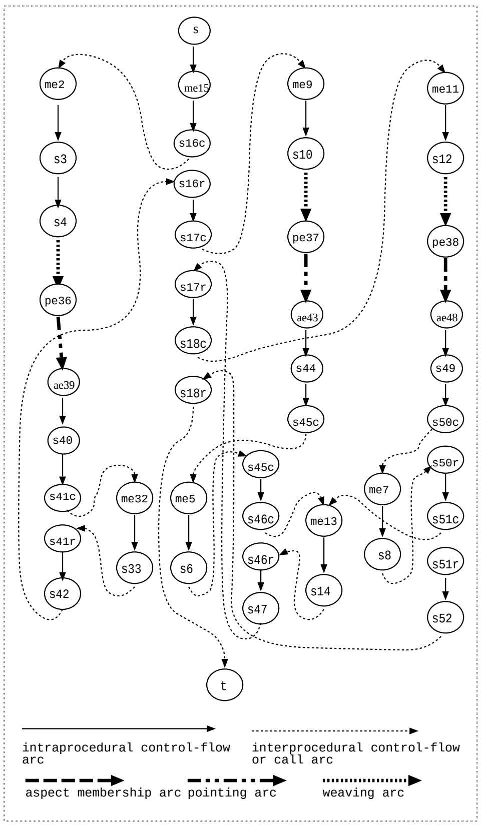

Figure 3: The SCFG for the program in Figure 1.

pectJ programs.

6. CONCLUSION

We have presented a regression test selection technique for AspectJ programs. Our technique is based on various types of control flow graphs that can be used to select from the original test suite test cases that execute changed code for the new version of an AspectJ program. Our code-based technique operates on control flow graphs of AspectJ programs. Our technique can be applied to modified individual aspects or classes as well as the whole programs that used modified aspects or classes. In 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.

7. REFERENCES

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

[2] R. T. Alexander, J. M. Bieman, and A. A. Andrews. Towards the Systematic Testing of Aspect-Oriented Programs. Technical Report CS-4-105, Department of Computer Science, Colorado State University, Fort Collins, Colorado, 2004.

[3] The AspectJ Team. The AspectJ Programming Guide. August 2004.

[4] 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.

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

[6] 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.

[7] 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.

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

[10] T. Koju, S. Takada, N. Doi. Regression Test Selection based on Intermediate Code for Virtual Machines. *Proc. International Conference on Software Maintenance*, pp.420-429, 2003.

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

[12] 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.

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

[14] 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.

[15] 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.

[16] S. Sinhai and M. J. Harrold. Analysis and Testing of Programs with Exception-Handling Construct. *IEEE Transactions on Software Engineering*, pp.849-871, September 2000.

[17] D. Sokenou and S. Herrmann. Aspects for Testing Aspects. *Workshop on Testing Aspect-Oriented Programs*, AOSD 2005, Chicago, USA, March 2005.

[18] T. Xie and J. Zhao. A Framework and Tool Supports for Generatiing Test Inputs of AspectJ Programs. *Proc. International Conference on Aspect-Oriented Software Development*, pp.190-201, Bonn, Germany, March 2006.

[19] D. Xu and W. Xu. State-Based Incremental Testing of Aspect-Oriented Programs. *Proc. International Conference on Aspect-Oriented Software Development*, pp.180-189, Bonn, Germany, March 2006.

[20] 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.

[21] 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.

[22] Y. Zhou, D. Richardson, and H. Ziv. Towards a practical approach to test aspect-oriented software. *Proc. 2004 Workshop on Testing Component-based Systems (TECOS2004)*, Net.ObjectDays, September 2004.