Towards the Systematic Testing of Aspect-Oriented Programs

Roger T. Alexander, **James M. Bieman**
Colorado State University
Department of Computer Science
Fort Collins, Colorado
{*rta,bieman*}@*cs.colostate.edu*

Anneliese A. Andrews
Washington State University
School of Electrical Engineering and Computer Science
Pullman, Washington
aandrews@eecs.wsu.edu

Abstract

The code that provides solutions to key software requirements, such as security and fault-tolerance, tends to be spread throughout (or cross-cut) the program modules that implement the "primary functionality" of a software system. Aspect-oriented programming is an emerging programming paradigm that supports implementing such cross-cutting requirements into named program units called "aspects". To construct a system as an aspect-oriented program (AOP), one develops code for primary functionality in traditional modules and code for cross-cutting functionality in aspect modules. Compiling and running an AOP requires that the aspect code be "woven" into the code. Although aspect-oriented programming supports the separation of concerns into named program units, explicit and implicit dependencies of both aspects and traditional modules will result in systems with new testing challenges, which include new sources for program faults. This paper introduces a candidate fault model, along with associated testing criteria, for AOPs based on interactions that are unique to AOPs. The paper also identifies key issues relevant to the systematic testing of AOPs.

1 Introduction

Aspect-Oriented Programming [11] is a new technology for dealing explicitly with separation of concerns in software development. Just as with the introduction of object-oriented programming, aspect-oriented programming brings a unique set of benefits and challenges. By far, the most compelling argument for aspect-oriented programming is that it significantly increases modularity and cohesion, thereby increasing understandability and easing the maintenance burden. In particular, AOPs use abstractions representing *concerns* that *cross-cut* the program modules that implement the primary functionality. For example, code that implements a particular security policy is commonly distributed across a set of classes and methods that are responsible for enforcing the policy. However, with AOPs, the code implementing the security policy can be factored out (typically into one aspect). This aspect localizes in one cohesive place the code that affects the implementation of multiple classes and methods [5, 6].

The use of AOPs alters the development process. Classes and methods of core concerns are developed and tested as before. However, instead of embedding the code for cross-cutting actions into method bodies, separate aspects are defined that contain the code. Later, the aspects are woven into the classes that represent the core concerns of the system. Once complete, the woven targets should be the composite of behavior of both core and cross-cutting concerns. The advantages of this approach include greater modularity and cohesion, a cleaner separation of concerns, and improved locality of change.

Regardless of the programming technology used, a primary development goal is the production of high quality software. Consequently, AOPs must also be tested and vetted for quality. Key questions related to quality include: *How do we adequately test aspect-oriented programs*? *How do we know that our testing and quality objectives have been attained*? *Is there some objective way to determine when we have tested enough*? These questions are not easy to answer because of the following issues:

- **Aspects do not have independent identity or existence**. They depend upon the context of some other class for their identity and execution context.
- **Aspect implementations can be tightly coupled to their woven context.** Aspects depend on the internal representation and implementation of classes into which they are woven. Changes to these classes will likely propagate to the aspects.
- **Control and data dependencies are not readily apparent from the source code of aspects or classes**. Due to the nature of the weaving process, the developer of classes or aspects knows neither the resulting control flow nor the data flow structure of the resulting woven artifact. Thus, relating failures to the corresponding faults may be difficult.
- **Emergent behavior**. The root cause of a fault may lie in the implementation of a class or an aspect, or it may be a side effect of a particular weave order of multiple aspects.

The above challenges cannot be addressed using traditional unit or integration testing techniques. Most unit level techniques depend upon the availability of source code, and, while these techniques are still applicable to classes that implement core concerns, they are not applicable to aspects. This is because aspects are not complete code units and their behavior often depends on the woven context. While test scaffolding is possible for simple aspects, it is not practically feasible for more realistic situations. Existing integration testing techniques focus on the interactions that occur between modules at their interfaces. In AOPs, integration is more fine grained and occurs with respect to the intra-method control and data flow. At this level, there are no well-defined interfaces.

Systematic testing of AOP systems must be based on fault models that reflect the structural and behavioral characteristics of aspect-oriented programs. Criteria and strategies for testing AOPs should be developed in terms of the fault model. In this paper we propose an initial fault model. We show how these fault models can be used to develop criteria and strategies for testing AOPs.

Section 2 explains how AOPs work and summarizes existing research on testing AOP code. Section 3 develops a fault model for AOPs. Section 4 uses this fault model to derive testing criteria. Section 5 draws conclusions and points out further work.

2 Background

The following sub-sections present basic concepts of aspect-oriented programming and briefly survey related work.

2.1 Aspect-oriented Programming Concepts

The core of a software-based system is the subject matter of some application domain. For example, in a library, the subject matter will include *Books*, *Periodicals*, library *Patrons*, the *Catalog* of works, and so on. A system designed to manage the operation of a library must incorporate each of these subject matter items and more. Collectively, the subject matter of the system form the *core concerns* of the system and constitute the domain information that must be managed. In the implementation of the system, each core concern will typically be represented as a single abstraction with a corresponding implementation using some concrete mechanism, such as a class in an object-oriented language. However, not all concerns can be represented in this discrete manner. Instead, the implementation of some concerns depend on the context provided by the behavior and concrete representation of other concerns. Such concerns are referred to as *cross-cutting concerns*.

A simple example from the library system is a policy concern that restricts the number and type of publications that a patron may have checked out at any given time. Behavior for enforcement of this policy spans several other core concerns, including *Books*, *Periodicals*, and *Patrons*. Each of these core concerns has a role to play in the enforcement of the policy concern. Consequently, rather than being encapsulated in a single location, the behavior of the policy concern is spread across the other participating concerns. In terms of conventional implementation using object-oriented (OO) or procedure-oriented languages, the code for the policy enforcement would be tangled with that of the code that implements the behavior of each participating core concern. Although necessary, this code entanglement reduces the cohesiveness of the core concerns, decreasing modularity and reducing the locality of changes.

A basic tenet of aspect-oriented programmng is that all concerns should be treated as modular units regardless of the limitations of the implementation languages. The primary mechanism for defining solutions to cross-cutting concerns is the *aspect*. Aspects encapsulate behavior and state of those cross-cutting concerns whose implementations must span across the core concerns that form the subject matter of a system. Aspects make it possible to create cohesive modules that implement specific cross-cutting concerns that otherwise would have to be distributed across many core concerns. By placing these cross-cutting concerns separately in an aspect, the core concerns are made more cohesive since their implementations are relieved of the burden of managing concepts unrelated to their purpose.

AspectJ is a language designed to support aspect-oriented programming in conjunction with the Java programming language. AspectJ achieves modularity with an aspect abstraction mechanism which encapsulates the behavior and state of a cross-cutting concern. Unlike classes, aspects cannot be directly instantiated as objects. The activation of an aspect depends upon context provided by the core concerns represented as classes. Thus, while cross-cutting concerns are expressed as separate modular units, their definition and execution depend upon the context of a core concern's control and data flow.

Figure 1 is an example of a simple aspect expressed in AspectJ [10]. The syntactic definition of an aspect consists of a set of patterns, referred to as *pointcuts*, that are used to select elements, called *join points*, that appear in the bodies of methods. These join points correspond to well defined locations in the control flow of a method of some concern. Examples of join points include method calls, method execution, field access, and object construction. Examples of pointcuts appear at lines 5 and 10 of Figure 1. Figure 2 shows a sample method having a *method call* join point at line 12, and its position in the method's control flow graph. Note that node 12 is control dependent on node 5.

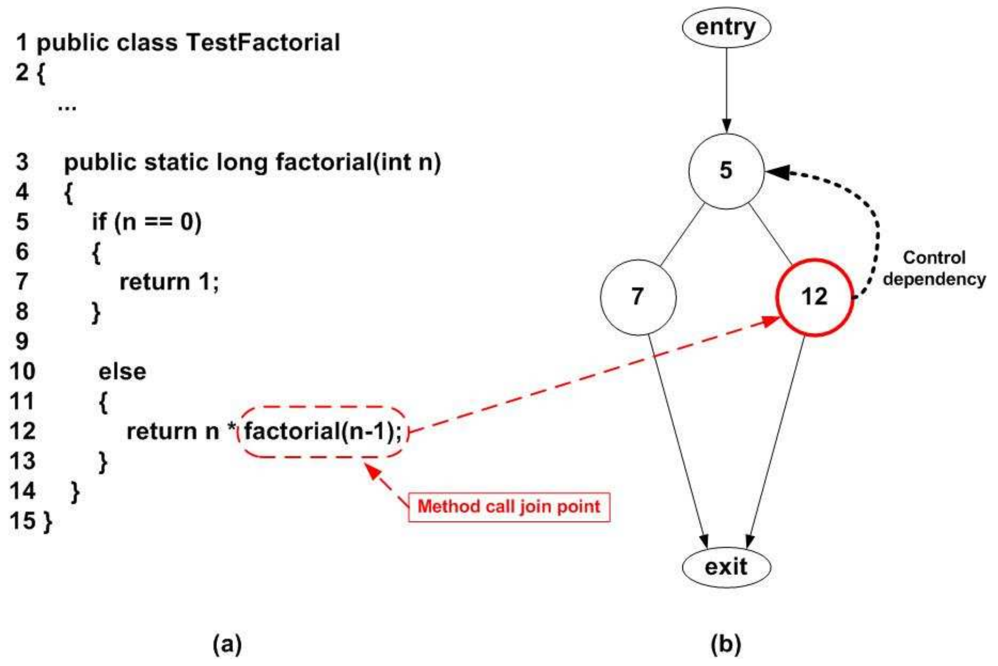
When a join point matches the pattern specified by an aspect's pointcut, *advice* is applied through a composition process called *weaving* to the matching join point in the method body. Advice is a method-like construct used to express the cross-cutting actions that must take place within the method body at the matched join point. It is the advice that is woven into the target concern. While advice is similar in notion to a method body, it is not a complete modular unit. Instead, advice generally represents a fragment of control and data that must be added to the body of an existing method [5].

Advice may take one of three forms: *before*, *after*, and *around*. *Before* advice consists of instructions that must execute in the method body before execution of the element of the matched join point. Similarly, *after* advice are instructions that must execute after the element corresponding to the matched join point has executed. Figure 3 (b) and (d) shows examples of *before* and *after* advice for the aspect shown in line 10 of Figure 1. Parts (a) and (c) of 3 depict the control flow schematic for the respective *before* and *after* advice. The schematic shows the control flow that weaving adds to the control flow graph of a method having one or more join points selected by the *topLevelFactorial-Operation* pointcut. The fragmentary nature of advice is depicted in 3 (a) and (c) as incomplete control flow graphs. Entry nodes are represented as a circle with a white center and exit nodes are reversed, having black centers. These circles represent where the advice will be connected to a method's control flow graph. The thick dashed nodes correspond to a method's control flow node for the join point matched by the pointcut.

Around advice consists of instructions that, after

```
1 public aspect OptimizeFactorialAspect
2 {
3    private Map _factorialCache = new HashMap();
4
5    pointcut factorialOperation(int n)
6       : call(long *.factorial(int))
7       && args(n)
8       ;
9
10    pointcut topLevelFactorialOperation(int n)
11       : factorialOperation(n)
12       && !cflowbelow(factorialOperation(int))
13       ;
14
15    before(int n) : topLevelFactorialOperation(n)
16    {
17       System.out.println("Seeking factorial for " + n);
18    }
19
20    long around(int n) : factorialOperation(n)
21    {
22       Object cachedValue = _factorialCache.get(new Integer(n));
23
24       if (cachedValue != null)
25       {
26          System.out.println("Found cached value for "
27                           + n
28                           + ": "
29                           + cachedValue)
30                           ;
31
32          return ((Long)cachedValue).longValue();
33       }
34
35       return proceed(n);
36    }
37
38    after(int n) returning(long result) : topLevelFactorialOperation(n)
39    {
40       _factorialCache.put(new Integer(n), new Long(result));
41    }
42 }
```

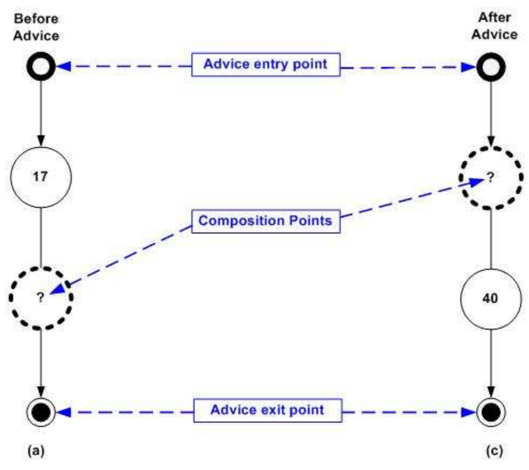
Figure 1. Sample aspect



```
1 public class TestFactorial
2 {
     ...

3    public static long factorial(int n)
4    {
5       if (n == 0)
6       {
7          return 1;
8       }
9
10       else
11       {
12          return n * factorial(n-1);
13       }
14    }
15 }
```

Figure 2. Sample method control flow graph



```
15   before(int n)
        : topLevelFactorialOperation(n)
16   {
17      System.out.println("Seeking factorial for " + n);
18   }
```
(b)

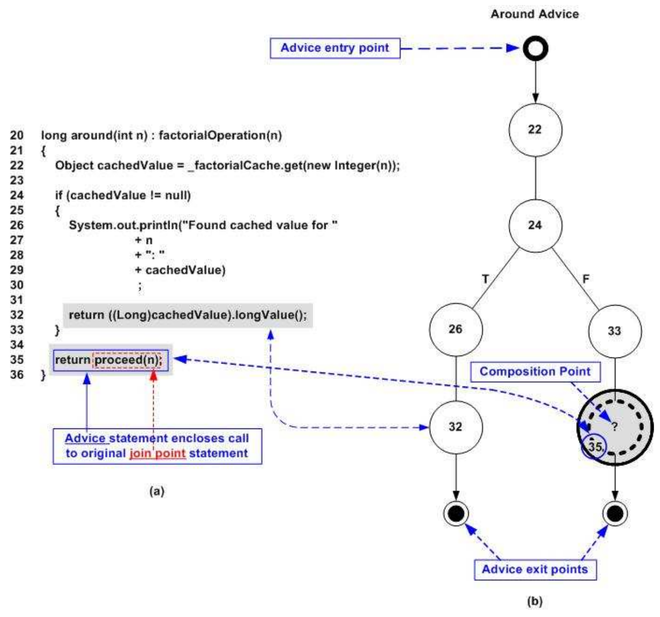
```
38   after(int n) returning(long result)
        : topLevelFactorialOperation(n)
39   {
40      _factorialCache.put(new Integer(n), new Long(result));
41   }
```
(d)

Figure 3. Sample *before* and *after* advice

weaving, surround matched join point elements and may alter the control flow and data dependencies of the method. *Around* advice can bypass the matched join point elements altogether, or make the matched join point's execution control dependent upon the *around* advice. Figure 4(a) shows the *around* advice for the aspect depicted in Figure 1 and Figure 4(b) gives control flow schematic for the advice. As the example shows, the around advice will add a new branch into the code of the core concern at the location of a matched join point. Also, the around advice adds an additional control flow edge (via node 32) and encapsulates the original join point as the value in a return statement (line 35). As this example shows, aspect-oriented programming has the possibility to make significant changes to the semantics of a core concern, thus creating new testing challenges.

2.2 Related Work

In aspect-oriented programming [12, 15], aspects capture functionality that cross-cuts code modules. Other research extends aspects and their use to the design level (e.g., [3, 7, 8, 9, 17, 20]. Research on testing software built via aspect-oriented programming is scarce. We conducted an informal survey of the



```
20   long around(int n) : factorialOperation(n)
21   {
22      Object cachedValue = _factorialCache.get(new Integer(n));
23
24      if (cachedValue != null)
25      {
26         System.out.println("Found cached value for "
27                      + n
28                      + ": "
29                      + cachedValue)
30                      ;
31
32         return ((Long)cachedValue).longValue();
33      }
34
35   return proceed(n);
36   }
```

Figure 4. Sample *around* advice

WWW, and citation databases such as Citeseer, ACM Digital Library, and the IEEE Computer Society's Digital Library. We were surprised to find very few researchers working on testing AOPs. It appears that the majority of work being done is on using aspect-oriented methods to build testing tools.

Zhao proposes an approach based on data-flow testing for unit testing AOPs [23]. This approach combines testing of individual aspects and classes that have the potential to be affected by one or more aspects. Definition-use pairs (du-pairs) are computed to determine what interactions between aspects and classes must be tested. These du-pairs are computed based on the ability of a variable definition in one module to reach a corresponding use in another module or within the same module.

Denaro and Monga use model checking to verify properties of aspects suitable for formal verification [4]. They do not consider the environment in which aspects will be used, but instead rely on the verified properties holding through system evolution. They illustrate their approach by verifying concurrency concerns for a set of aspects. A similar approach is used by Ubayashi and Tamai [21].

An approach based on three-valued model checking is presented by Li, et al. [14]. Their approach allows for reasoning about features and interactions that occur as the result of composition (i.e. weaving).

While one, of course, can use existing black-box and white box testing strategies, if either functional specifications or code are available, this paper focuses on testing the unique relationships between modules due to aspect-oriented programming. It would be more effective and efficient to use testing techniques that focus on testing the results of weaving aspects into code representing primary concerns. Testing methods specifically tailored to aspect-oriented programming depend on a model of potential sources of faults that can occur when using AOPs.

3 AOP Fault Model

A fault model for AOPs should reflect those characteristics of AOPs that are distinct from object-oriented and procedure-oriented programming. A key issue is identifying the fault types that are unique to AOPs.

Because aspect-oriented programming is a new technology, new fault types have not been identified from practical experience. However, our background in developing testing and analysis techniques for object-oriented programs, along with a fault model, suggests to us that aspect-oriented programming introduces new types of faults [18, 1].

Fault models for AOPs must be based on the nature of faults and failures in AOPs and the unique characteristics that make testing AOPs challenging.

3.1 The nature of faults and failures in AOPs

When a failure occurs, the first challenge is to diagnose the failure and detect the fault. For non-aspect-oriented programs, one examines the code and possibly instruments it with probes to isolate and localize a fault. Dealing with failures in aspect-oriented programs requires a similar approach. However, to detect a fault in an AOP, the code of the woven aspects must also be examined. There are four potential sources of faults in a program with woven aspects:

1. **The fault resides in a portion of the core concern that is not affected by a woven aspect**. The fault is peculiar to the primary abstraction and could occur if there was no weaving.

2. **The fault resides in code that is specific to the aspect, isolated from the woven context**. Such a fault would be present in any composition that included the aspect. However, the fault resides in aspect code that is independent of the data and control dependencies induced by the weaving process.

3. **The fault is an emergent property created by interactions between one aspect and the primary abstraction.** Such faults would occur when weaving introduces new data or control dependences that were not present in the primary abstraction or the aspect alone. These new dependencies arise from the integration and interaction of code and data between the primary abstraction and the aspect.

4. **The fault is an emergent property created when more than one aspect is woven into the primary abstraction**. The difficulty of locating

the source of faults is compounded when multiple aspects are involved. The fault may arise only with a particular permutation of aspects with respect to the primary abstraction.

The testing effort is likely to be much greater whenever faults involve an aspect.

3.2 Essential differences between AOPs and object-oriented programs

AOPs are similar to object-oriented programs in that both have classes, interfaces, methods, packages, etc. Obviously, any Java program is a degenerate case of an AspectJ program that has no aspects. Thus, the types of faults that can occur in a Java program can also occur in an AspectJ program; the source of these faults are the same as those in alternative 1 (above). New types of faults can occur as a result of using the aspect-oriented features of an aspect-oriented programming language. In addition, aspect weaving can affect the types of faults commonly found in object-oriented programs (OOPs) and procedural programs. One way to identify potential sources of faults in AOPs is to examine what a programmer must do to develop an AOP.

Obviously, an AOP developer must consider additional concepts beyond those present in OOPs. The computational model of an AOP is based on a dynamic call graph resulting from method invocations. Within the execution that the call graph represents are join points where advice has been applied based on a pointcut match. To specify the patterns in a pointcut, the developer must analyze the syntactic structure of a core concern to determine those join points that must participate in the cross-cutting behavior supplied by an aspect. Based on the syntactic structure present in those join points, the aspect developer must specify an appropriate set of pointcuts and advice that, after weaving, will result in the desired behavior.

In some cases, the aspect developer must consider a wider context beyond that provided at a particular join point. This occurs, for example, when an aspect must maintain some information on a per-object basis. A common example is a cache-management aspect that must track the last access time of each object stored in the cache [13].

Another consideration is the flow of control within a dynamic call graph with respect to a particular join point. In certain situations, information must be collected only when the flow of control is executing within a body of code corresponding to a particular join point, such as a call to a method *m*. In this example, information is collected from the thread of execution until control returns to the caller of *m*. In other cases, information collection may be restricted to when the thread of execution has not yet returned to the caller of *m* **and** is not executing within the body of *m*.

In addition to execution, an aspect developer must consider what information about a join point must be passed to advice. In AspectJ, some of this information is made available automatically. For example, the predefined pointcuts *this*(), *target*(), and *args*() are used to collect information about the context of the join point. Such information includes arguments to method calls and constructor invocations, a reference to the target object, and return values.

Clearly, the unique characteristics of AOPs do not occur in object-oriented, or even procedure-oriented, programs. Each characteristic has the possibility to manifest new fault types that ultimately lead to program failures.

3.3 A candidate fault model for AOPs

The candidate fault model is based on the peculiarities of AOPs. It is defined through careful analysis of the unique constructs of AspectJ and reflects an initial evaluation of classes of potential faults.

3.3.1 Incorrect strength in pointcut patterns

Pointcuts contain specifications that select join points of a particular type according to a signature that includes a pattern. For a pointcut p, each matching join point j of a concern C will have the advice associated with p woven into j. The strength of the pattern in the signature of p determines which join points are selected. If the pattern is too strong, some necessary join points will not be selected. If the pattern is too weak, additional join points will be selected that should be ignored. Either case is likely to cause incorrect behavior of the woven target. The statements in the woven advice and the statements that are executed after the woven join point determine whether a pattern strength error will introduce a fault. Infections, invalid state caused by a program fault [22], induced by woven advice could be subsequently canceled by coincidental correctness.

3.3.2 Incorrect aspect precedence

The order in which advice from multiple aspects are woven into a concern affects system behavior, especially when there are mutual interactions between aspects through state variables in the core concern. In AspectJ, weave order is determined by the specification of aspect precedence. For example, the aspect with the highest precedence executes its *before* advice on a join point prior to executing the *before* advice of the lower precedence aspect. If precedence is not specified, the order in which advice is applied remains undefined [2]. Precedence is of no concern as long as the effects of woven advice are mutually independent.

3.3.3 Failure to establish expected postconditions

The clients of core concerns expect those concerns to behave according to their contracts. A client has the responsibility to ensure that a method precondition holds prior to calling the method. Given that the precondition is ensured, the client can reasonably assume that the method's postcondition will be satisfied. This is a basic requirement for behavioral inheritance (i.e. subtyping) [16]. Clients expect method postconditions to be satisfied regardless of whether or not aspects are woven into the concern. Hence the behavioral contracts of the concern should hold after the weaving process. Thus, for correct behavior, woven advice must allow methods in core concerns to satisfy their postconditions. Defining advice that does not cause behavioral contracts to be broken in all likely weave contexts and with all likely combinations of aspects will be a difficult challenge for aspect developers, and a likely source of errors.

3.3.4 Failure to preserve state invariants

A concern's behavior is defined in terms of a physical representation of its state, and methods that act on that state. In addition to establishing their postconditions, methods must also ensure that state invariants are satisfied. Ensuring that weaving does not cause violations of state invariants is another difficult challenge for aspect developers, and another source of errors.

3.3.5 Incorrect focus of control flow

A pointcut designator selects which of a method's join points to capture. This selection is determined at weave time. However, there are often cases where the information needed to correctly make such a decision is available only at run time. Sometimes join points should only be selected in a particular execution context. This context could be within the control structure of a particular object, or within the control flow that occurs below a point in the execution. Consider the *topLevelFactorialOperation* pointcut shown in lines 15-18 of Figure 1. In this example, the join points are selected when *factorialOperation* is called, but not as a recursive call during the execution of a currently executing call to *factorialOperation*. The expression `!cflowbelow(factorialOperation(int))` in *topLevelFactorialOperation*'s pointcut designator restricts the selection of join points. Failure to restrict execution to the proper context could result in a failure that is likely to be difficult to diagnose.

3.3.6 Incorrect changes in control dependencies

Figure 5 shows the logical control flow resulting from weaving the *around* advice given in Figure 1 to the method in Figure 2.[1] The grey region highlights the control flow resulting from the around advice. Control flow nodes and edges that denote the *around* advice are represented as thick edges. Thin edges and nodes are from the control flow graph of the method. The node corresponding to the matched join point, node 12, is depicted with a thick dashed circle.

Careful examination of the resulting control flow graph reveals a change in control dependencies. In Figure 2, node 12 is control dependent upon the decision at node 5, whereas in Figure 5, node 12 is now control dependent upon node 24. This change is due to the branch that occurs at node 24. This example makes clear that around advice can significantly alter the behavioral semantics of a method. A similar ar-

[1]The actual control flow may be different depending upon the implementation of the weaver. For example, in AspectJ, advice is often woven in as a method call [2].

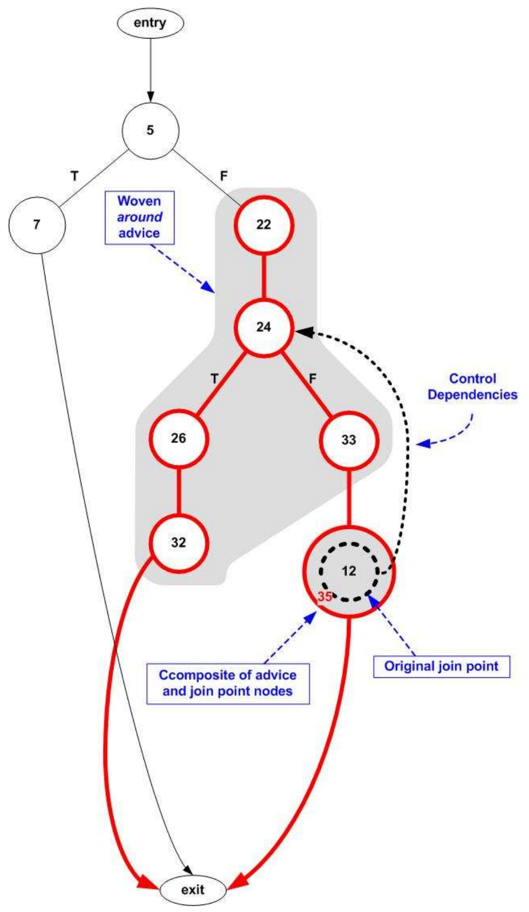


Figure 5. Changes on control flow after weaving *around* advice

gument can be made for changes in data dependencies (i.e., du-pairs).

One of our primary goals is to develop a detailed and thorough understanding of how faults and failures occur in AOPs. As a first step, we defined a candidate fault model based on reasoning about how AOPs work. The fault model will likely need to be extended and adjusted in the future.

4 Proposed Testing Criteria

Testing criteria are derived from the fault model. We assume for now a strategy that requires that all core concerns have been tested before weaving and are deemed relatively fault free with respect to some testing criteria. The criteria specify additional testing requirements.

Testing criteria focus on the behavior of the core concern, the behavior of each aspect, and the behavior of the weaver. The following are brief descriptions of proposed criteria for each of the types of faults described in Section 3.3:

1. Incorrect strength in pointcut patterns: These faults can cause the aspects to fail, rather than the core functionality. Thus, a test of the aspect is required here.

2. Incorrect aspect precedence: These faults will occur when multiple aspects interact, and are affected by the weave order. Testing all weave orders should reveal these faults.

3. Failure to establish postconditions: These faults can cause core concern methods to fail. A reasonable criteria is to re-test all methods that have code weaving using the original test set.

4. Failure to preserve state invariants: These faults also can cause core concern methods to fail. Retesting of concern methods is also appropriate.

5. Incorrect focus of control flow: These faults can cause advice to activate at the wrong time. A criteria to reveal these faults may be a form of condition coverage of point cut designators.

6. Incorrect changes in control dependencies: These faults will affect core concern behavior, like those of fault types 3 and 4.

Testing strategies and criteria for fault types 3, 4, and 6, which involve aspects that cause core concerns to fail, will be similar to regression testing strategies and criteria. Fault types 1 and 2 involve incorrect weaving — aspects are woven into the wrong core concerns. Fault type 5 involves errors in the run time behavior of an aspect.

5 Conclusion

The complexity of software development motivated aspect-oriented programming as an approach to separate concerns and to build high quality software more easily, with higher maintainability, and a better chance at successful evolution. We believe that aspect-oriented programming has tremendous potential for building the software of the future, but only if we can pair effective and efficient testing techniques with aspect-oriented programming software construction methods. Thus we see the combination of sound AOP construction methods and systematic AOP testing as an important step in broad acceptance of aspect-oriented programming as a software development approach.

As a first step, we developed a candidate fault model for AOPs and derived testing criteria from the candidate fault model. This does not yet constitute a fully developed testing approach. First, the candidate fault model needs to be empirically validated and possibly extended and refined. We have an ongoing study to validate and further define this fault model. Second, answers will have to be found to the following open questions:

1. **Are there ways to test aspects on their own?** One approach might be to use the same coverage criteria on the woven target that was used to test the core concern. However, this is difficult since the weave process of many aspect-oriented programming languages weave directly to byte code [19, 11]. We could analyze the byte code, except that it will be difficult to refer back to the source code of the core concern and/or the aspects. Also, compiler optimizations increase the difficulty. Further, the weaving process can produce byte code that has no direct connection to any available source artifact (i.e., core concerns or aspects).

2. **Can we reverse engineer the weave process?** The goal is to work back through the weave steps in order to make a connection to the woven source artifacts.

3. **Can we measure test coverage after weaving?** We need to understand the types of coverage metrics that can be reliability collected about individual aspects from the execution of a woven target.

4. **How do we test aspects that interact with a core concern?** The execution of many aspects, logging for example, does do not have side-effects on the concern that they are woven into. However, the execution of other aspects, such as object persistence, does have an effect. Developers need to be able to determine how a core concern is affected by the execution of aspects for both testing and maintenance activities.

5. **How do we test aspects that mutually interfere?** Many aspects can be woven together, and it is possible for the aspects to interact in ways that may not be apparent by examination of their source.

6. **How do we test aspects whose effects must span more than one concern?** For example, consider authentication policy enforcement, where an authentication aspect will be woven into class *C*. All clients of *C* must also be modified to account for the new postcondition induced by an authentication failure. Such a policy could cause a chaining of effects; clients of the clients of *C* would also require modification and so on.

We believe that our work is a first important step in developing an effective approach to the systematic testing of AOPs.

References

[1] R. T. Alexander, J. Offutt, and J. Bieman. Syntactic fault patterns in OO programs. In *Proceedings of Eighth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '02)*, Greenbelt, Maryland, December 2002.

[2] T. AspectJ. The AspectJ(TM) Programming Guide, 2002.

[3] S. Clarke and J. Murphy. Developing a tool to support the application of aspect-oriented programming principles to the design phase. In *Proceedings. of the International Conference on Software Engineering (ICSE'98)*, Kyoto, Japan, April 1998.

[4] G. Denaro and M. Monga. An experience on verification of aspect properties. In *Proceedings of the 4th international workshop on Principles of software evolution*, pages 186–189. ACM Press, 2002.

[5] T. Elrad, M. Aksits, G. Kiczales, K. Lieberherr, and H. Ossher. Discussing aspects of AOP. *Communications of the ACM*, 44(10):33–38, 2001.

[6] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming: Introduction. *Communications of the ACM*, 44(10):29–32, 2001.

[7] J. L. Fiadeiro and A. Lopez. Algebraic semantics of co-ordination or what is it in a signature? In *Procs. of the 7th International Conference on Algebraic Methodologyand Software Technology (AMAST'98)*, volume 1548 of *Lecture Notes in Computer Science*, pages 293–307, Amazonia, Brazil, January 1998. Springer Verlag.

[8] R. B. France, I. Ray, G. Georg, and S. Ghosh. An aspect-oriented approach to design modeling. *IEE Proceedings – Software, Special Issue on Early Aspects:Aspect-Oriented Requirements Engineering and Architecture Design*, to appear, 2004.

[9] J. Gray, T. Bapty, S. Neema, and J. Tuck. Handling crosscutting constraints in domain-specific modeling. *Communications of the ACM*, 44(10):87–93, October 2001.

[10] M. Kerstern. Demo: Aspectj: The language and development tools, November 4-8 2003.

[11] G. Kiczales, E. Hilsdale, , J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An Overview of AspectJ. In *15th European Conference on Object-Oriented Programming*, Budapest, Hungary, 2001.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, and W. G. G. J. Palm. Getting started with aspectj. *Communications of the ACM*, 44(10):59–65, October 2001.

[13] R. Laddad. *AspectJ in Action*. Manning Publications, Inc., 2003.

[14] H. Li, S. Krishnamurthi, and K. Fisler. Verifying cross-cutting features as open systems. In *Proceedings of the tenth ACM SIGSOFT symposium on Foundations of software engineering*, pages 89–98. ACM Press, 2002.

[15] K. Lieberherr, D. Orelans, and J. Ovlinger. Aspect-oriented programming with adaptive methods. *Communications of the ACM*, 44(10):39–41, October 2001.

[16] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.

[17] P. Netinant, T. Elrad, and M. E. Fayad. A layered approach to building aspect-oriented systems. *Communications of the ACM*, 44(10):83–85, October 2001.

[18] J. Offutt, R. Alexander, Y. Wu, Q. Xiao, and C. Hutchinson. A fault model for subtype inheritance and polymorphism. In *Proceedings of the 12th International Symposium on Software Reliability Engineering*, pages 84–93, Hong Kong China, November 2001. IEEE Computer Society Press.

[19] H. Ossher and P. Tarr. Hyper/j: multi-dimensional separation of concerns for java. In *Proceedings of the 22nd international conference on Software engineering*, pages 734–737. ACM Press, 2000.

[20] J. A. D. Pace and M. R. Campo. Analyzing the role of aspects in software design. *Communications of the ACM*, 44(10):66–73, 2001.

[21] N. Ubayashi and T. Tamai. Aspect-oriented programming with model checking. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 148–154. ACM Press, 2002.

[22] J. M. Voas. Pie: A dynamic failure-based technique. *IEEE Trans. Software Engineering*, 18(8):717–727, August 1992.

[23] J. Zhao. Data-flow-based unit testing of aspect-oriented programs. In *Proc. 27th Annual IEEE International Computer Software and Applications Conference (COMPSAC'2003)*, pages 188–197, Dallas, Texas, December 2003.