

A State-Based Approach to Testing Aspect-Oriented Programs

Dianxiang Xu, Weifeng Xu, and Kendall Nygard

*Department of Computer Science
North Dakota State University
Fargo, ND 58105, USA*

{dianxiang.xu, weifeng.xu, kendall.nygard}@ndsu.edu

Abstract

This paper presents a state-based approach to testing aspect-oriented programs. Aspectual state models, as an extension to the testable FREE state model of classes, are exploited to capture the impact of aspects on the state models of classes. To generate test suites for adequately testing object behavior and interaction between classes and aspects in terms of message sequences, we transform an aspectual state model to a transition tree, where each path from the root to some leaf node indicates a template of test cases, i.e. message sequences. Since the state-based approach is directly built upon the test design patterns for object-oriented programs, it is not only applicable to the simultaneous development of classes and aspects, but also to the incremental development of aspects based on the existing classes.

Keywords: Aspect-oriented programming, software testing, state model, transition tree

1. Introduction

An aspect-oriented program typically consists of a number of modules (or classes) and aspects that can be woven into an executable whole [1,2,3]. The crosscutting mechanism of aspects frees the programmer from interweaving different concerns (i.e. goals, concepts, or areas of interests) in a monotonous program hierarchy imposed by the base language. This greatly facilitates identifying and modularizing separate concerns that crosscut multiple functional components or objects. The dynamic behavior of objects, including interactions, dependencies, and constraints on the message sequences, is therefore determined collectively by the specification of both objects (classes) and aspects. The interaction between aspects and classes may introduce a variety of bug hazards into the system [4]. To name a few, improper *join points*, *pointcuts*, and *advices* likely lead to unexpected system behaviors or even failures. In Aspect-Oriented Software Development (AOSD), validating

whether or not the aspects of crosscutting concerns are implemented correctly is a major issue. It requires adequate exercise of classes as well as aspects.

Generally speaking, the aspects of crosscutting concerns alter the control flows defined in the core concern. From the perspective of state models of system behavior, they not only modify the state-transition relations but also possibly introduce extra states in the state models of objects defined by their classes. This can affect or even change the behavior of objects specified by the base programs. Therefore, a state-based testing method for object-oriented programs would be insufficient for adequately testing aspect-oriented system implementation.

Inspired by Binder's work on the FREE (Flattened Regular Expression)-based test design patterns for object-oriented programs [5], this paper presents a state-based approach for testing aspect-oriented programs. The motivations are twofold: (1) Considering that FREE has been applied successfully to a variety of test design patterns for object-oriented programs, adapting the FREE-based testing approach would likely support two AOSD paradigms – simultaneous development of classes and aspects and incremental development of aspects based on existing classes. Support of incremental testing for the later is particularly useful for maintaining and enhancing a legacy system using AOSD. (2) The FREE-based testing approach for aspect-oriented programs is expected to be applicable to the common aspect-oriented modeling methods that have been developed or are being developed by the AOSD community. Specifically, UML has been extended to support aspect-oriented modeling [6,7,8,9]. FREE is similar to the state model in UML although it has restrictions and definitions not found in the UML. Nevertheless, a UML state model that follows the FREE conventions is testable.

In the state-based approach to testing aspect-oriented programs, we first extend FREE to aspectual state models (ASM) for specifying both classes of the core concern and aspects of the crosscutting concerns, then transform an

ASM to a transition tree, which implies a test suite for adequately testing object behavior and interaction between classes and aspects in terms of message sequences.

The rest of the paper is organized as follows. Section 2 briefly reviews related work on testing aspect-oriented programs. In section 3, we introduce aspectual state models as a high-level abstraction of the core concern (classes) and crosscutting concerns (aspects). Section 4 presents the transition tree – based method for testing object behaviors and interaction of classes and aspects. Section 5 concludes the paper.

2. Related Work

AOSD as an emerging paradigm of software development is still in its infancy. It is not surprising to see that little research on testing aspect-oriented programs has been published [4].

Zhao has proposed a data flow based approach to unit testing of aspect-oriented programs [10]. For each aspect or class, the approach performs three levels of testing, i.e., intra-module, inter-module, and intra-aspect/intra-class testing. Definition-Use pairs (DU-pairs) are constructed to determine what interactions between aspects and classes must be tested. Zhao and Rinard [11] have also exploited system dependence graphs to capture the additional structures in aspect-oriented features such as *join points*, *advice*, *aspects*, and interactions between *aspects* and classes. In this approach, control flow graphs are constructed at both system and module level, and test suites are derived from control flow graphs. No fault model is targeted to help detect most likely faults.

Alexander, Bieman, and Andrews [4] have recently proposed a fault model for aspect-oriented programming, which includes six types of faults: incorrect strength in pointcut patterns, incorrect aspect precedence, failure to establish postconditions, failure to preserve state invariants, incorrect focus of control flow, and incorrect changes in control dependencies. While this fault model has not yet constituted a fully-developed testing approach, it is undoubtedly useful for developing testing tools and determining coverage strategies and criteria.

To reduce the cost of testing aspects, Zhou, Richardson and Ziv [18] have recently introduced a control flow based approach to selecting relevant test cases for testing aspects. A tool has been developed to calculate test coverage and select relevant test case if new test cases should be developed when reused test cases can not cover the aspects under test satisfactorily.

Ubayashi and Tamai [12] have proposed a model checking method for verifying whether or not an aspect-oriented program satisfies expected properties. A similar method is presented by Denaro and Monga [13]. This

work primarily concerns with the properties of concurrency, such as deadlock, liveness, and fairness. Li, Krishnamurthi and Fisler’s three-valued model checking approach allows reasoning about interactions as the result of weaving [14]. Different from verification, Sereni and Moor [15] have proposed a method for static analysis of aspects based on a syntactic model of pointcut designators using regular expressions.

It is worth mentioning that aspects can provide a convenient way to develop testing tools or built-in tests [16], although the work along this line is in essence irrelevant to the paper. Nevertheless, testing such aspects is also a critical issue.

3. Aspectual State Models

Aspectual State Model (ASM) is an extension to the FREE state model for Object-Oriented Software Development (OOSD). FREE uses the UML interpretation of Statecharts with the testability extensions. Objects are encapsulated entities of data and operations that can receive messages from and send messages to other objects [17]. The interactions, dependencies, and constraints on the message sequences determine the behavior of objects. A FREE model depicts the states and dynamic behaviors of objects and provides guidelines for implementation. It can be used at both system and class levels.

For example, Fig. 1 is the FREE model for class *Account*. For the purposes of illustration, it is a simplified version of the *Account* class in [5], but will be extended to deal with the overdraft concern in the aspect-oriented paradigm. This will facilitate discussing the difference between AOSD and OOSD. Here, the states of an *Account* object include *Open*, *Frozen*, *Inactive* and *Closed*, and the guarded (conditional) transitions or visible methods include *open*, *debit*, *credit*, *balance*, *freeze*, *unfreeze*, *close*, *settle*, etc. The state-transition relations determine the possible state changes of an *Account* object. The interface of the *Account* class is given in Fig. 2.

Modeling aspect-oriented structures requires three modeling elements: base elements, crosscutting elements, and crosscutting relationships. The base element in our approach is the FREE model. As to the crosscutting elements and relationships, we introduce some additional notation for the basic AOP constructs - *join points*, *pointcuts* and *advices*. For simplicity, other AOP notions such as introduction, aspect inheritance, and aspect composition will not be discussed in this paper. The primary new notation is shown in Fig. 3. We use solid arrows and boxes for guarded transitions and states in the FREE model of a class, and dashed arrows and boxes for the crosscutting mechanism (*join points*, *advices*, etc.) and new states introduced by *aspects*. For example, S_1 , S_2 , S_3 , S_4 , and S_5 in solid boxes in Fig. 3 are states in the

corresponding FREE model of a class, and S_6 is a new state introduced by the crosscutting concern. The solid arrow from S_1 to S_2 with guarded transition $m_1[cond_1]$ means m_1 , provided that condition $cond_1$ is satisfied, transforms state S_1 into state S_2 in the FREE model. A dashed arrow with a solid diamond attached to the other end indicates a *join point* at the entry or exit of transition $m_1[cond_1]$ (a *join point* with some *before* or *after* advice). It is an entry (or exit) *join point* for $m_1[cond_1]$ if the diamond is near the beginning (or end) of the arrow for transition $m_1[cond_1]$. For example, $m_1[cond_3]$ means the *advice* for the entry *join point* of m_1 transforms S_1 to S_6 if $cond_3$ evaluates true. A dashed arrow with a solid bullet at the other end means a *join point* with some *around* advice. An 'X' following the bullet indicates the original transition no longer applies.

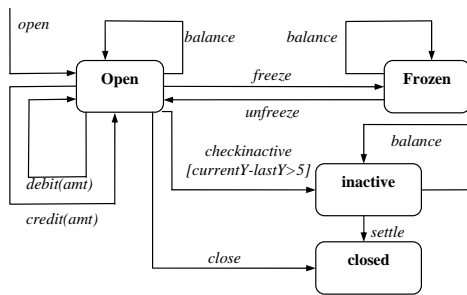


Figure 1. The FREE model for class Account

```

public class Account{
    public double open()
    public Money balance()
    public void credit(double creditAmt)
    public void debit(double debitAmt)
    public void freeze()
    public unfreeze()
    public double settle()
    public void close()
}

```

Figure 2. The interface of class Account

An ASM is a state model that is based on FREE and the new notation. An *aspect* may introduce new states and transitions into the state model of its base class. Again, let us consider the previous *Account* example. Suppose the requirements for class *Account* have changed as follows:

- Overdrawing activities must be identified to accommodate new policies, such as financial penalties. A negative balance indicates an *overdrawn* state.
- Dealing with the concern of “debit legality” is necessary (The original class *Account* does not involve itself in the distinction between legal and illegal withdrawal). The legality issue needs to be handled before a *debit* transaction is actually performed. A limited overdraft is allowed. But an account will be frozen if the amount of an

attempted overdraft exceeds a given threshold (say, MAX_OVERDRAFT, a negative value).

- Inactiveness of accounts needs to be adjusted. An account is inactive if there are no transactions within five years (the original condition) and the balance is less than a given threshold, namely MIN_BALANCE (the new condition).

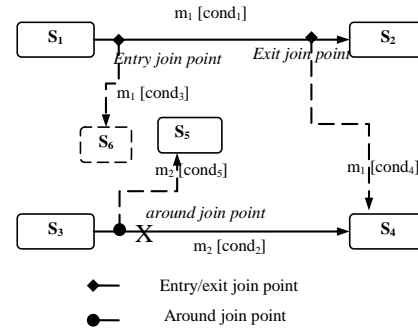


Figure 3. Basic notation for ASM

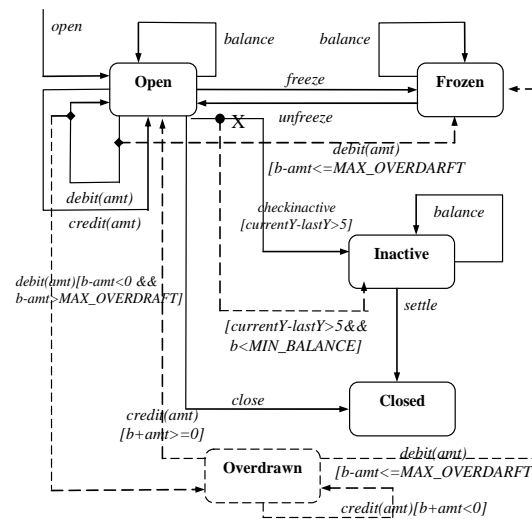


Figure 4. The ASM for Account

Note that the above requirements may be elicited together with those for the previous *Account* class, but here they are treated as a separate concern. Referring to them as changes of requirements facilitates comparing the methods for testing object-oriented programs and testing aspect-oriented programs. Also, the changes of business rules can be addressed by the AOP paradigm without the need to make direct change to the *Account* class. Fig. 4 shows the ASM model of the AOP solution, where *Overdrawn* is a new abstract state and *b* refers to the current balance. For method *debit(amt)* at the *Open* state, the entry *join point* indicates that the new state is *Overdrawn* if $b-amt < 0 \ \&\& \ b-amt > MAX_OVERDRAFT$; the exit *join point* indicates the new state is frozen if *b*-

amt>*MAX_OVERDRAFT* (i.e. an attempt to withdraw too much money). Method *credit(amt)* at the *Overdrawn* state may either retain the state or transform the state to *Open*. In the original FREE model of *Account*, *Overdrawn* is not recognized, but hidden in the *Open* state. An *Open* account may become *Overdrawn* due to a *debit* transaction that makes the balance negative, yet within the threshold. For simplicity, methods *debit(amt)* and *balance* at the *Overdrawn* state are not considered.

The AOP implementation outline for the above changes of requirements is given in Fig. 5. The changes are addressed by *before*, *after* and *around advices*, respectively.

```
public aspect OverdraftControl {
    pointcut entryCheck(Account account):
        call(void debit(double))
        || target(account);
    pointcut exitCheck(Account account):
        call(void debit(double))
        && call(void credit(double))
        && target(account);
    pointcut checkInactiveness(Account account):
        call(void active(double))
        && target(account);

    before(Account account):
        entryCheck(account)
    {...} //before advice

    after(Account account):
        exitCheck(account)
    {...} // after advice

    Object around(Account account):
        checkInactiveness (account)
    {...} // around advice
}
```

Figure 5. The *OverdraftControl* aspect

Normally, a base class (e.g. *Account*) is unable to be aware of the existence of extra states (e.g. *Overdrawn*) introduced by an *aspect* (e.g. *OverdraftControl*) although the behavior of the base class is dynamically affected or even changed by the crosscutting elements at runtime. In addition, the base class alone does not recognize changes to the state-transition relations. From the testing perspective, therefore, we must exercise the extra states and changes of state-transition relations imposed by *aspects*.

Of course, one could implement the aforementioned changes of requirements for *Account* by redesigning or extending the original *Account* class without introducing AOP. Suppose the new account class is named *NewAccount*. The FREE model of *NewAccount* would be slightly from the ASM in Fig. 4 because of the absence of crosscutting notation. This model would include the *Overdraw* state and the associated transitions, though. From the perspective of state models, the FREE model of

NewAccount is more readable than the ASM in Fig.4. The former primarily supports modeling from scratch, whereas the latter supports incremental modeling for changes of requirements and separation of concerns. The ASM is advantageous in two aspects: 1) It can better guide AOP implementation; and 2) It is more effective for adequately testing aspect implementation, as will be discussed later.

ASM is developed to provide a testable model of class behaviors along with additional *advices* defined in *aspects*, which are dynamically attached when specific *join points* are reached. While the most important implication of *aspects* is that they provide a flexible mechanism for modularizing concerns that crosscut multiple classes, we have treated *aspects* from the perspective of individual classes. For the purposes of testing, this does not lose generality.

4. Transition Tree-Based Testing

This section discusses a testing method that generates test cases directly from the ASM of an aspect-oriented design. To deal with crosscutting concerns, the testing method extends the transition tree based testing for object-oriented programs. The purpose of the method is to validate dynamic behavior of objects, including interactions, dependencies, and constraints on the message sequences in terms of the state model. Undoubtedly, an aspect-oriented implementation can fail to realize the design in the way much like an object-oriented implementation does. In particular, the interactions between classes and *aspects* can cause failures because of improper use of *join points*, *pointcuts* and *advices* for the separate concerns.

The fundamental idea of the transition tree based testing is to transform a state model to a transition tree. In the transition tree, each path from the root to a terminal leaf node, i.e. a sequence of transitions (method invocations or messages), is a test requirement for testing object behaviors. The test requirement becomes a concrete test case if the variables are assigned specific values that satisfy the corresponding conditions. The following algorithm outlines the steps for generating the transition tree from an ASM:

- Step 1:* The initial state of the ASM is the root node of the transition tree.
- Step 2:* For each non-terminal leaf node in the transition tree, draw an edge and new node for each resultant state in the ASM. The edge represents the event that transforms the object's state in the leaf node to the resultant state.
- Step 3:* If the transition is picked up by a *join point*, add the *join point* to the edge at a corresponding location.

Step 4: If the state represented by the new node already occurs in another node of the transition tree or is a final state in the ASM, the node is marked terminal – it will no longer be expanded.

Step 5: Repeat steps 2, 3 and 4 until all leaf nodes are marked terminal.

For example, Fig. 6 shows the transition tree generated for the ASM in Fig. 4 using *Open* as the initial state. For comparison, we use a dashed box (e.g. *Overdrawn*) to represent a state introduced by the *aspect* of crosscutting concern (i.e. not present in the FREE model of the base class). A path with dashed arrows is a path that does not occur in the transition tree generated from the FREE model of the base class. It indicates a test requirement that is specific to the crosscutting concern. For example, the path from *Open* → *Overdrawn* → *Overdrawn* implies a message sequence *open()*, *debit(amt1)*, and *credit(amt2)*, where $b - amt1 < 0$ and $b - amt1 \geq MAX_OVERDRAFT$ and $b - amt1 + amt2 < 0$, and b is the initial balance. With specific values assigned to b , $amt1$, and $amt2$, the message sequence is a test case for exercising the new requirements. To adequately test the *aspect*, each of the aforementioned paths should be exercised at least once.

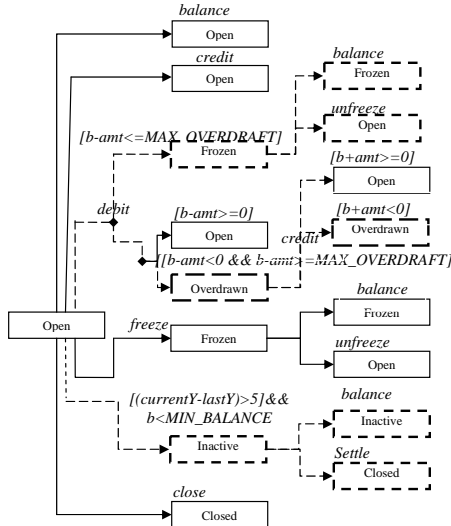


Figure 6. Transition tree for the ASM in Fig. 4

The above test cases are essentially for the purposes of testing the intended behaviors of *aspects*. We need to test unintended behaviors of the *aspects* ('dirty tests' for *aspects*) - what will happen if those constraints associated with the transitions are not met. The previous algorithm has only considered sending messages that meet the conditions for the transitions to fire. It is necessary to analyze and identify additional test cases for each transition condition. Here we apply the multi-conditional coverage. Some conditions have been covered by the transition tree mentioned above. For instance, all

conditions for the transition *debit* from the *Open* state have been included. Table 1 shows the transition conditions for the checkinactive method. According to conditions that are not fully covered in the previous transition tree, we can further expand the transition tree to a conditional one. For example, the multi-conditional coverage for $(currentY - lastY) > 5 \ \&\& \ b < MIN_BALANCE$ is included in the expanded transition tree in Fig. 7.

Table 1: Transition conditions

Transition	Condition	Next State
Check inactive	$(currentY - lastY) \leq 5 \ \&\& \ b < MIN_BALANCE$	Open
Check inactive	$(currentY - lastY) \leq 5 \ \&\& \ b \geq MIN_BALANCE$	Open
Check inactive	$(currentY - lastY) > 5 \ \&\& \ b < MIN_BALANCE$	Inactive
Check inactive	$(currentY - lastY) > 5 \ \&\& \ b \geq MIN_BALANCE$	Open

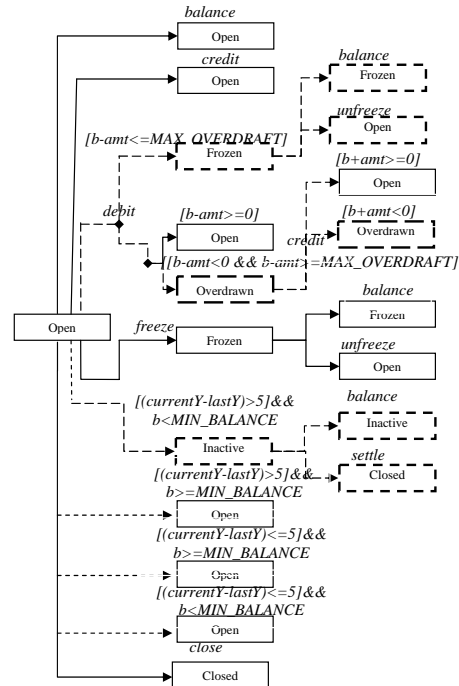


Figure 7. Expanded transition tree

To adequately test an aspect-oriented implementation for the ASM in Fig. 4, we may exercise the program through all the paths in Fig.7. Such a test suite can achieve N+ coverage, which will reveal all state control faults, all sneak paths, and many corrupt state bugs, like the transition tree-based testing for object-oriented programs [5]. In addition, the test suite can also reveal some faults that are specific to *aspects*. Such faults include incorrect strength of *pointcut* patterns and failure to preserve state invariants, which are two types of faults

in the AOP fault model proposed by Alexander et al [4]. For example, a weaker (or stronger) *pointcut* pattern picks up more (fewer) *join points* than it is supposed to. This type of faults can be detected by the above transition tree based testing because the transition tree indicates a test suite for all the expected *join points*. If the implementation misses an expected *join point*, the test cases for the expected *join point* will reveal the error. If the implementation has an extra *join point*, the dirty tests of the expected *join points* will detect it. Of course, the incorrect results of such an error must be observable, which is a common fault detection requirement of testing.

Note that, if the *aspects* are viewed incremental to the base classes that are already tested adequately with their transition trees, there is no need to repeat the tests. We only need to exercise those cases that are solely introduced for the crosscutting concern. The transition-tree based testing method therefore provides a sound support for AOSD without negative effects on the development process.

5. Conclusions

We have presented the state-based testing approach. The approach allows for reuse of the test cases designed for the base programs and is therefore consistent with the standpoint of ‘programming by difference’ - aspects are essentially incremental to object-oriented programs, which facilitates clean separation of concerns by constructing new programming components and specifying how they differ from existing components [19]. The extended behavior expressed in the new components can be plugged in without requiring modification or duplication of existing code.

The basis of the testing approach in this paper, ASM, is a preliminary extension to the FREE state model for the identification and specification of basic crosscutting notions. Since a state-based approach often suffers from the state explosion problem, it is worth discussing how to identify the paths that are of most interest or importance. Another issue is how the testing approach fits in other UML-based modeling methods for AOSD, such as [6][7][8]. Finally, we expect to extend the approach for revealing the likely faults in aspect composition, i.e. composition of crosscutting concerns.

Acknowledgments

This work was supported in part by the NSF under grant EPS-0132289.

References

1. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J. M. Loingtier, and J. Irwin. Aspect-oriented

- programming. In *Proc. of the European Conference on Object-Oriented Programming (ECOOP'97)*, LNCS 1241, pp. 220-242.
2. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W.G. Griswold. An overview of AspectJ. In *Proc. of ECOOP'01*, pp. 327-353.
3. J. Gradecki and N. Lesiecki. *Mastering AspectJ: Aspect-Oriented Programming in Java*. Wiley, 2003.
4. R. T. Alexander, J. M. Bieman, and A.A. Andrews. Towards the systematic testing of aspect-oriented programs, *Technical Report*, Colorado State University. <http://www.cs.colostate.edu/~rta/publications/CS-04-105.pdf>.
5. R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 2000.
6. C. Chavez and C. Lucena, A Metamodel for aspect-oriented modeling, *The Workshop on Aspect-Oriented Modeling with UML*, 2002.
7. T. Aldawud, and A. Bader, UML profile for aspect-oriented software development, *The Third International Workshop on Aspect Oriented Modeling*, 2003.
8. S. J. Mellor, A framework for aspect-oriented modeling. *The 4th AOSD Modeling With UML Workshop*, 2003.
9. I. Ray, R. France, N. Li, and G. Georg. An aspect-based approach to modeling access control concerns. *Information and Software Technology*, vol. 46, no.9, pp. 575-587, 2004.
10. J. Zhao, Data-flow-based unit testing of aspect-oriented programs, In *Proc of the 27th Annual IEEE International Computer Software and Applications Conference (COMPSAC'03)*, pp.188-197, 2003.
11. J. Zhao and M. Rinard, System dependence graph construction for aspect-oriented programs, *MIT-LCS-TR-891*, Laboratory for Computer Science, MIT, March 2003.
12. N. Ubayashi and T. Tamai. Aspect-oriented programming with model checking. In *Proceedings of the 1st International Conference on Aspect-oriented Software Development*. April 2002.
13. G. Denaro and M. Monga. An experience on verification of aspect properties. In *Proceedings of the 4th International Workshop on Principles of Software Evolution*, pp. 186-189. ACM Press, 2002.
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*, pp. 89-98. ACM Press, 2002.
15. D. Sereni and O. de Moor. Static analysis of aspects. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*. March 2003.
16. J. M. Bruel, J. Araújo, A. Moreira, and A. Royer: Using aspects to develop built-in tests for components, *The 4th AOSD Modeling with UML Workshop*, 2003.
17. B. Meyer. *Object-Oriented Software Construction*. Englewood Cliffs, N.J.: Prentice-Hall, 1988.
18. Y. Zhou, D. Richardson, and H. Ziv. Towards a practical approach to test aspect-oriented software. In *Proc. 2004 Workshop on Testing Component-Based Systems (TECOS 2004)*, Sept. 2004.
19. D. Orleans, Incremental programming with extensible decisions, In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*, April 2002, The Netherlands.