

A Model-Based Approach to Test Generation for Aspect-Oriented Programs

Weifeng Xu

Department of Computer Science
North Dakota State University
Fargo, ND 58105, U.S.A
weifeng.xu@ndsu.edu

Dianxiang Xu

Department of Computer Science
North Dakota State University
Fargo, ND 58105, U.S.A
dianxiang.xu@ndsu.edu

ABSTRACT

This paper presents an approach to generating tests for adequately exercising interaction between aspects and classes based on aspect-oriented UML models. In this approach, an aspect-oriented model consists of class diagrams, aspect diagrams, and sequence diagrams. Since methods of classes and advices of aspects are both specified by sequence diagrams, the advices on a given method can be weaved into a new sequence diagram, which is essential to the weaving mechanism in AOSD. For a woven sequence diagram, the approach exploits goal-directed reasoning to construct a flow graph for a given coverage criteria (e.g. the combination of branch coverage and polymorphic coverage in this paper), and further expands the graph to a flow tree, where each path is a test. Then the approach derives objects with desired states for exercising each test in the flow tree.

Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging - *Testing tools (e.g., data generators, coverage testing)*

General Terms

Algorithms, Design, Languages

Keywords

Aspect-oriented programming, testing, UML, sequence diagram, test generation.

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. To test an aspect-oriented system implementation, it is desirable to adequately exercise classes as well as aspects.

This paper presents an approach to generating tests for adequately exercising interaction between aspects and classes based on aspect-oriented UML models. In this approach, an aspect-oriented model consists of class diagrams, aspect diagrams, and sequence diagrams. Since methods of classes and advices of aspects are both specified by sequence diagrams, the advices on a given method can be weaved into a new sequence diagram, which is essential to the weaving mechanism in AOSD. Based on a woven sequence diagram, the approach exploits goal-directed reasoning to construct a flow graph for a given coverage criteria (e.g. the combination of branch coverage and polymorphic coverage in this paper), and further expands the graph to a flow tree, where each path is a test. Then the approach derives objects with desired states for exercising each test in the flow tree.

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 UML-based aspect-oriented modeling. In section 4, we present how to generate test cases. 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

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, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Submission to AOSD'05 Workshop on Testing AOP.

Copyright XXXX XXX X-XXXXX-XXX-X/XX/XXXX...\$X.00.

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.

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 concurrence, such as deadlock, liveness, and fairness. Li, Krishnamurthi and Fislser’s three-valued model checking approach allows for 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.

To reduce the cost of testing aspects, Zhou, Richardson and Ziv [25] 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.

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. ASPECT-ORIENTED MODELING

Based on the work [17][18][19] [20], we extend UML for aspect-oriented modeling. Our goal is to allow aspect-oriented models to carry structural and behavioral information that is necessary for test automation.

As an extension to UML class diagrams, an aspectual class diagram represents a collection of declarative (static) model elements, such as classes, aspects, types, and their contents and relationships. Using AspectJ meta model [21], crosscutting concerns can be expressed in Fig. 1, where each aspect has two components: pointcut and advice. Port is a structural feature of a classifier that specifies a distinct interaction point between that classifier and its environment or between the classifier and its internal parts. Connector specifies a group of links that enables communication between two or more instances. A port represents a join point and a connector represents a pointcut expression.

Fig. 2 shows an aspectual class diagram with two classes: *Rental* and *Movie*. Each class has a join point collected in pointcut expression *fieldAccess*. *fieldAccess*, encapsulated in aspect *AccessControl*, facilitates communication between *Rental* and *Movie* objects.

We use sequence diagrams to model interaction between objects, including both class methods and aspect advices. Since sequence diagrams capture messages flows, they facilitate weaving aspects into classes. In the following, we introduce some additional notation for the weaving, upon which the test generation approach is built.

In sequence diagrams, we represent join points as links. Field references or field assignments are stereotyped as “pseudo” invocations of “pseudo” operations, such as *get()* and *set()*, as in [22]. There are no links to represent execution and initialization

join points because a preceding link does represent two or three join points. Fig. 3 illustrates two types of join points, i.e. method call and method execution. A message *m(x)*, denoted by a link, represents both types. The call designator is used when one is interested in the actual calling of the method or constructor as opposed to the execution of code within a join point. The duration when a control passes through these join points are denoted as dotted rectangles. In the example, left rectangle indicates the method call duration, and the right rectangle indicates the duration of method execution. Solid diamonds are used to distinguish different type of advices access points. The diamond attached to the top of the rectangle is denoted as a *before* type access point, and the diamond located at the bottom of the rectangle indicates the access point of *after* type. The solid circle at the top of the rectangle is the *around* type access point, and it should be placed after *before* type. To represent *around* advices, a pseudo control object *around* is introduced to play a role of delegation.

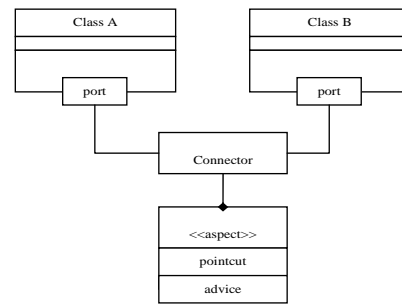


Figure 1. Aspectual class diagram

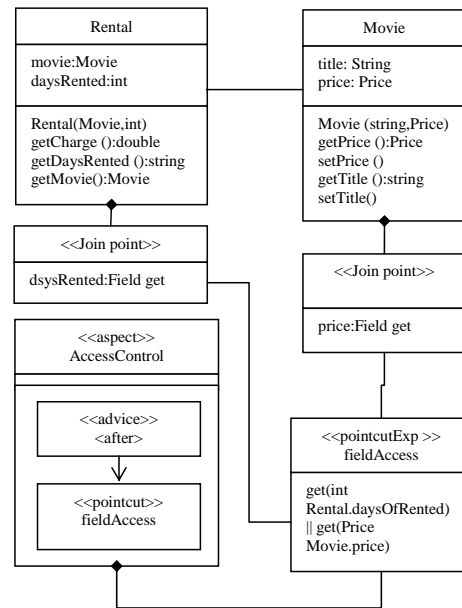


Figure 2 MovieRental aspectual diagram

Take Fig. 4 as an example. The sequence diagram, describing the behavior of *getCharge*, consists of three objects, *Rental*, *Movie* and *Price*, and five messages, *getPrice*, *getPregisteredPrice*, *getPeriod*, *getDiscountedPrice*, *chargeWithinPeriod* and *chargeBeyondPeriod*. The basic business rules for calculating the total charge are as follows: 1) All the movies have been

categorized as Regular movies, Children’s movies and New released movies; 2) For each type of the movies, registered, period and discounted prices are pre-defined; and 3) A basic amount, named registered price, will be charged within the period. Discounted rates are applied to each extra day. The objects are represented as rectangles with the underlined class name within the rectangle, and the interactions (messages) between different objects in the sequence diagram are denoted by directed arrows. Note that a method is private if an object invokes itself, for instance, *chargeWithinPeriod()* and *chargeBeyondPeriod()* are private methods.

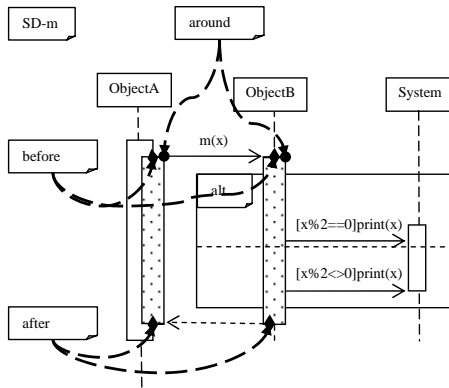


Figure 3 Advice access point types

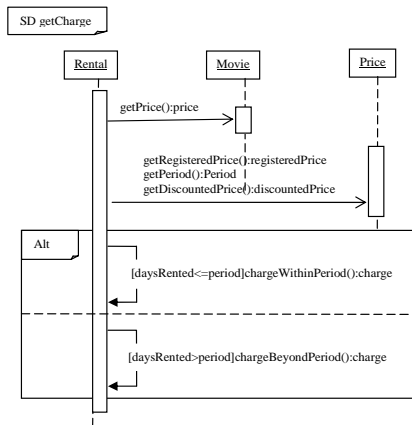


Figure 4 Sequence diagram for *getCharge()* use case

Now suppose the requirements for class *Movie* have been changed to deal with the security concern. For example, the original class *Movie* does not involve itself in the distinction between legal and illegal access. The legality issue needs to be handled before a method is actually performed - only legal access is allowed. Let us just use the most simplistic mechanism for security check, i.e. password-based access control. We assume that *price* is a password protected field in the class *Movie*. Note that multiple methods and multiple fields in each class may be involved in the security concern. For simplicity, only one field of the class is shown here.

The AOP implementation for the above change of requirements is given in Fig. 5. The change is implemented by an *around* advice.

```
public aspect AccessControl {
    private boolean authenticated;
    public pointcut controlMethods(): get(price Movie.price);
```

```
Object around() : controlMethods () {
    if (authenticated) {
        return proceed();
    }
    else {
        LoginScreen loginScreen = new LoginScreen();
        loginScreen.setVisible(true);

        // The authentication procedure
        // checks if it is someone we know of
        if((loginScreen.getUsername().equals("Frank")) &&
            new String(loginScreen.getPassword()).equals("pswd"))
        { authenticated = true;
          loginScreen.dispose();
          return proceed();
        }
        loginScreen.dispose();
        return null;
    }
}
```

Figure 5 AccessControl Aspect

Fig. 6 shows the woven diagram for *getCharge* with the security concern. Normally, a base class (e.g. *Movie*) is unable to be aware of the existence of extra interactions and constrains (e.g. *authorized=false*) introduced by an *aspect* (e.g. *AccessControl*) 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 message sequence. From the testing perspective, therefore, we must exercise the extra messages and changes of the ordering of message sequences imposed by *aspects*.

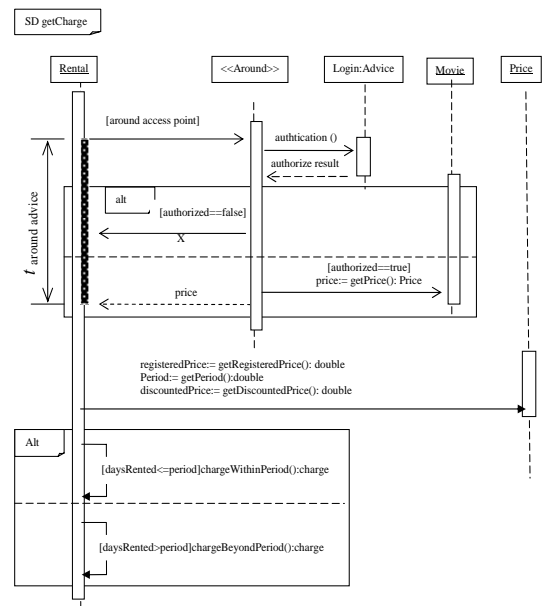


Figure 6 The woven diagram for *getCharge* use case

4. MODEL-BASED TEST GENERATION

Based on a woven sequence diagram, we first generate a goal-directed flow graph in a bottom-up way, then we expand the flow graph to a flow tree. The flow graph generation starts from the end of the sequence diagram. The successors of each node depend on the given coverage criteria (i.e. polymorphic and branch coverage in this paper). In the flow tree, each path from a leaf to the root represents one run of the woven sequence diagram. In

order to make the run possible, necessary data and objects are generated by reasoning about the reverse path, i.e. from the root to the leaf.

4.1 FLOW GRAPH GENERATION

In our approach, flow graphs constructed from sequence diagrams provide a testable model of class and aspect behaviors. Formally, a flow graph is a directed acyclic graph which is defined as a pair (V, E) , where V is a set of vertices, and E is a set of edges between the vertices $E = \{(u,v) \mid u, v \in V\}$. Each vertex is a message expression together with the condition for reaching the point. If a solid circle is attached to a message, the message has a crosscutting concern with an around advice (e.g. 4a, 4b, and 4c in Fig.7, which is the outline flow graph for *getcharge* in Fig.6).

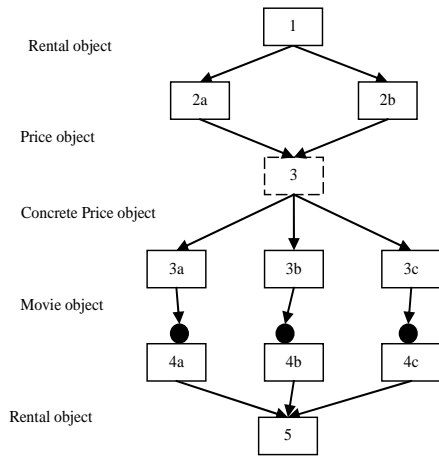


Figure 7. Flow graph outline for *getcharge*

Generally, a goal-directed flow graph is generated by three steps: First, we simplify the sequence diagram so that only the first-level of message invocations are used. For example, *getcharge* method in *Rental* object invokes methods in *Movie* and *Price* objects; however, from the perspective of a *Customer* object, it only interacts with a *Rental* object. If an investigation of how *getcharge* works is needed, objects of *Movie* and *Price* are required to be considered. Instead of using return messages to indicate the response of message sending, the sending message also includes the expected return value and type of the value unless the message is a join point, a start message and return message. Second, we associate each message in the simplified sequence diagram with its source and destination object.

- Creating a node for each message m in the sequence diagram.
- Identifying m 's source object o_1 and destination object o_2 , and labeling them as source: o_1 and dest: o_2 .
- Finding the condition under which the message is sent, and labeling the condition.
- Filling the upper rectangle with the message expression.

Third, we traverse the sequence diagram. The edges are directed to reflect the ordering of the messages and required objects to fulfill corresponding received message.

- The starting node is the return (e.g. *return charge* for *getCharge()*).
- Each condition statement is represented by an *Alt* in UML 2.0. An *Alt* splits the requested message into sub-request based upon the condition (i.e. branch coverage).

For example, *return charge* requests *chargeWithPeriod()* and *chargeBeyondPeriod()* if *daysRented* is greater than the default period.

- If an object of an abstract class is requested (i.e. the message is polymorphic), the node is expanded to include each concrete subclass of the abstract class. In the example, *Price* is expanded to three concrete classes *RegularPrice*, *NewReleasedPrice* and *ChildrenPrice*.
- For each loop block, the condition of the loop can be expressed similarly.

Fig. 8 shows the flow graph derived from the woven sequence diagram in Fig. 6. Branch coverage for $daysRented \leq period$ and polymorphic situations for *price* are considered.

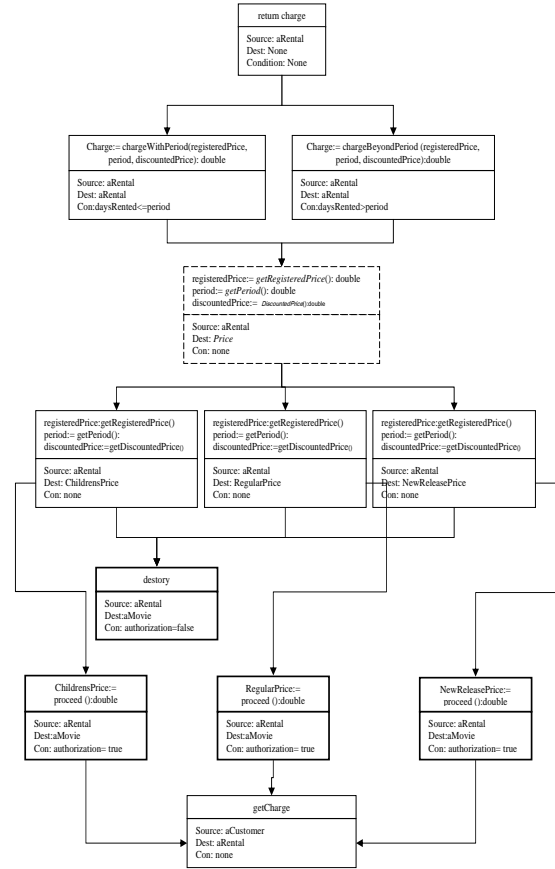


Figure 8 Flow graph for *getCharge*

4.2 FLOW TREES

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 *advice*s for the separate concerns.

We further transform a goal-directed flow graph to a flow tree, where each path from a leaf node to the root (i.e. a sequence of requested messages or method invocations) indicates a test case. The following outlines the algorithm:

- Step 1:* The initial message is the root node.
- Step 2:* For each non-terminal leaf node, draw an edge and new node for each requested non-abstract

message. The edge represents the request event that transforms the object's received message in the leaf node to the requested message to process the received message. If the message is abstract, ignore the message and the edge should connect to the each concrete message, respectively.

Step 3: For each node with multiple incoming edges in the flow graph, duplicate the node in the flow tree so that no more than one edge is connected to the same vertex.

Step 4: If the transition is picked up by an access point, add a new edge and a new node for each return message from the corresponding advice.

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

For example, Fig. 9 shows the flow tree for the flow graph in Fig.7. Each path describes a message sequence. For comparison, a return message from advices is denoted by a thicker rectangle. Note that although all the terminal nodes 6 are the same message, the states of the object are different.

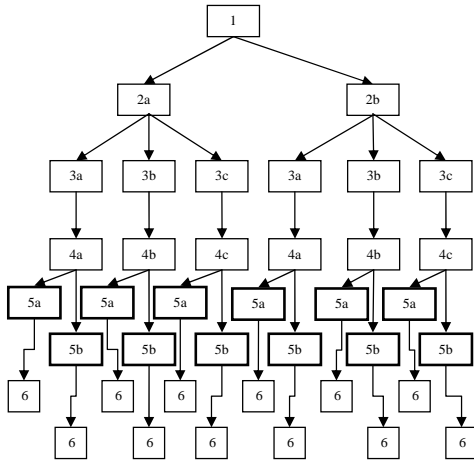


Figure 9 Flow tree for *getcharge*

Let's consider how to generate the test data for a particular path. Fig. 10 shows how test data is generated for path 1-2a-3a-4a-5a-6. One of our goals is to initialize state for an object (e.g. *Rental*). The request has been claimed at the beginning of a request sequence. Based on the message information and aspectual diagram in Figure 2, the construction of a *Rental* object needs two input parameters, a *Movie* object and a *daysRented* filed. The creation of a new *Movie* object can be deferred for the next request. However, all the primitive type variables can be initialized when they are needed. Specifically, boundary value analysis and equivalence partitioning may be applied to choosing values for primitive types. For example, we assign 2 to *daysRented*, 2 to *registeredPrice*, 3 to *period*, and 1.5 to *discountedPrice*. The assignments also satisfy message firing conditions, such as *daysRented* ≤ *period*. Next, an object of *ChildrensPrice* is instantiated based on the same rules. A *Movie* object can be created after the *ChildrensPrice* has been created. Finally, with the *Movie* object in hand, a *Rental* object is able to be initialized to exercise the path.

The algorithm for generating the test data for a particular path is outlined as follows:

Step 1. If it is not a final node of the path, identifying the new object O_1 needs to be created and add the O_1 to an incomplete object list l_1 if O_1 is not in the list.

Step 2. If it is a final node but there is object in the list l_1 , assign the last object as the new object O_1 .

Step 3. For each input parameter of the constructor of O_1 .

- If the input parameter is primitive type, assign an appropriate input value to the variable based on the requirement specification and required conditions.
- If the input parameter is another new object O_2 which has not been created yet, push O_2 to the incomplete object list l_1 .

Step 4. If all the input parameters are assigned values or the O_2 can be found in the completed list l_2 , delete O_1 from l_1 and add it to l_2 which is completed object list.

Step 5. For each input parameter for the methods of the object that to be created.

- If the input parameter is primitive type, assign an appropriate input value to the variable based on the requirement specification and required conditions.
- If the input parameter is another new object O_2 , add O_2 to the incompleted object list l_1 .

Step 6. Goto step 1

Step 7. Last object in the l is the resultant initialized object.

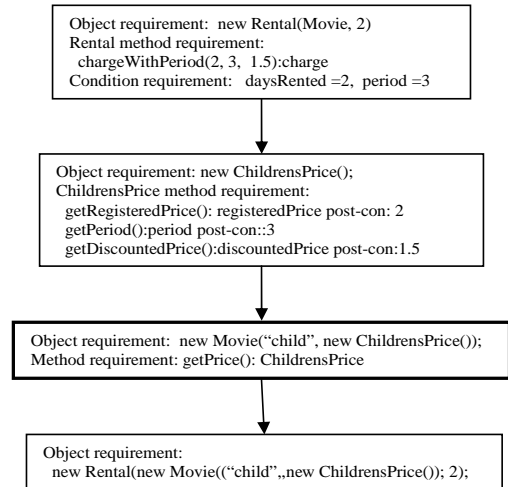


Figure 10 Test data generation

5. CONCLUSION

We have presented the model-based approach to generating tests for aspect-oriented programs that can achieve a given coverage criteria. Flow graphs for sequence diagrams of class methods and aspect advices provide a testable basis for aspect-oriented programs. Specifically, techniques can be developed to address a variety of testing questions. Examples include:

- How can we design test cases to exercise both base elements and crosscutting elements without modifying the test cases originally designed for base elements?
- Which additional test cases need to be generated to adequately exercise crosscutting elements?

- How to mock objects and aspects to carry out the tests, especially when some classes or aspects are not implemented yet?

This paper is a preliminary report of our ongoing research on model-based testing of aspect-oriented programs. Since the test generation approach applies to both base programs and aspects, it facilitates identifying the difference between the test suite for the base program and the test suite for the woven program (particularly when the aspects are mostly *before* and *after* advices). This difference indicates to what extent the former is reusable for the latter. It is important because aspects are in essence incremental to base programs.

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 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 ECOOP01*, 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, Position paper for the *Workshop on Aspect-Oriented Modeling with UML*, 2002.
- [7]. T. Aldawud, and A. Bader, UML profile for aspect-oriented software development, Position paper for 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, *Proceedings of the 27th Annual IEEE International Computer Software and Applications Conference (COMPSAC'2003)*, pp.188-197, Dallas, Texas, USA, November 3-6, 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]. W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. In N. Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 24, pages 433-444, New York, NY, 1989. ACM Press.
- [19]. D. Orleans, Incremental programming with extensible decisions, In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*, April 22-26, 2002, Enschede, The Netherlands.
- [20]. Zavaleta, Fuster and Morillo: An approach to Aspect Modeling with UML 2.0, *The 5th AOSD Modeling with UML Workshop*, 2004.
- [21]. Yan Han, Günter Kniesel and Armin B. Cremers: A Meta Model and Modeling Notation for AspectJ, *The 5th AOSD Modeling with UML Workshop*, 2004.
- [22]. D. Stein, S. Hanenberg, and R. Unland. An UML-based aspect-oriented design notation for aspectJ. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 106–112. ACM Press, 2002.
- [23]. O. Pilskalns, A. Andrews, R. France, S. Ghosh (2003), "Rigorous Testing by Merging Structural and Behavioral UML Representations", *Sixth International Conference on the Unified Modeling Language*, 2003, San Francisco, CA, USA, October 20-24, 2003.
- [24]. Mackinnon T., Freeman S., Craig P. Endo-Testing: Unit Testing with Mock Objects. In *proceedings of the eXtreme Programming and Flexible Processes in Software Engineering – XP 2000*. 2000.
- [25]. 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.