Testing Aspect-Oriented Programs with UML Design Models

Dianxiang Xu, Weifeng Xu
Department of Computer Science
North Dakota State University
Fargo, ND 58105, U.S.A
{dianxiang.xu, weifeng.xu}@ndsu.edu

W. Eric Wong
Department of Computer Science
University of Texas at Dallas
Richardson, TX 75803, U.S.A
ewong@utdallas.edu

Abstract
The new constructs in aspect-oriented programming bring new types of programming faults with respect to crosscutting concerns, such as incorrect pointcuts and advice. This paper presents a UML-based approach to testing whether or not an aspect-oriented program conforms to its expected crosscutting behavior. We explore aspect-oriented UML design models to derive tests for exercising interactions between aspects and classes. Each aspect-oriented model consists of class diagrams, aspect diagrams, and sequence diagrams. For a method under test, we weave the sequence diagrams of the advice on the method into the method’s sequence diagram. Based on the woven sequence diagram and class/aspect diagrams, we then generate an AOF (Aspect-Object Flow) tree by applying coverage criteria such as condition coverage, polymorphic coverage, and loop coverage to woven sequence diagrams. In the AOF tree, each path from the root to a leaf is an abstract message sequence, indicating a template of test cases. A concrete test case is obtained by creating objects that satisfy the collective constraints in the template. Our empirical study shows that the model-based testing approach is capable of revealing several types of aspect-specific faults, including incorrect advice type, incorrect (weaker or stronger) pointcut strengths, and incorrect aspect precedence.

Keywords
Software testing, model-based testing, aspect-oriented modeling, aspect-oriented programming, UML, test generation.

1. Introduction
Defects and failures persist in computer software. A major reason for this is the inherent complexity of software. Hierarchical decomposition in traditional software development methodologies is yet ineffective for dealing with many concerns (e.g., access control, persistence, concurrence control, etc.) that have a crosscutting impact on the components of a system (Kiczales et al., 1997; Jacobson and Ng, 2005). As a new avenue to reduce complexity, aspect-oriented programming (AOP) provides a paradigm for separating and modularizing such crosscutting concerns (Kiczales et al., 1997; Kiczales et al., 2001). The weaving mechanism of AOP automatically integrates primary concerns (i.e., classes) and crosscutting concerns (i.e., aspects) into an executable whole. It frees the
developers from interweaving primary and crosscutting concerns. The new constructs of AOP, however, yield new types of programming faults with respect to the crosscutting concerns, such as incorrect pointcuts and advice (Alexander et al., 2004). Revealing these faults requires exercising the interactions between aspects and their base classes.

This paper presents a model-based approach to testing whether or not an aspect-oriented program conforms to its expected crosscutting behavior specified by aspect-oriented UML design models. An aspect-oriented model consists of class/aspect diagrams and sequence diagrams that describe the static structure and dynamic behavior, respectively, of an aspect-oriented program. To derive tests for exercising interactions between aspects and classes, we first weave the sequence diagrams for the methods of classes and the advice of aspects and then generate an Aspect-Object Flow (AOF) tree from the woven sequence diagram. To construct an AOF tree, we apply coverage criteria such as condition coverage, polymorphic coverage, and loop coverage to the woven sequence diagram. These coverage criteria account for the situations that affect the execution of aspects (e.g., whether or not aspects are triggered). For example, the condition coverage ensures that both the true and false branches of a conditional behavior are exercised; the polymorphic coverage indicates that a polymorphic message affected by aspects is exercised with each subclass of the class in the polymorphic message. In an AOF tree, each path from the root to a leaf is an expected sequence of messages between objects and aspects. For each path, we derive concrete test cases by creating objects that satisfy the collective constraints along the path. To deal with situations where the exact oracle value of a test is difficult to define, we treat the expected message sequence of a test as the oracle and use the aspect-oriented paradigm to trace the actual messages during program execution. Specifically, using a generic trace aspect, the test harness automatically generates a corresponding trace aspect for each test so as to determine whether the test passes or fails.

The rest of this paper is organized as follows. Section 2 reviews related work. Section 3 is an overview of the UML-based approach to testing aspect-oriented programs. Section 4 introduces aspect-oriented modeling with UML. Section 5 describes test generation and execution. Section 6 presents our empirical study. Section 7 concludes the paper.

2. Related Work

Our approach is related to testing of aspect-oriented programs, model-based testing, and aspect-oriented modeling. This section briefly reviews the related work.

2.1 Testing of Aspect-Oriented Programs

While AOP provides a flexible mechanism for modularizing crosscutting concerns, it raises new challenges for testing aspect-oriented programs. Alexander et al. have 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 (Alexander et al., 2004). This fault model has not yet constituted a fully-developed testing approach. While some faults (e.g., incorrect pointcut strength and incorrect aspect precedence) are undoubtedly useful for developing testing tools and determining coverage strategies, others are subtle. For example, failures to establish postconditions or preserve state invariants assume that the contract of classes should be enforced by aspects at the design level.

Zhao has proposed a data flow-based approach to unit testing of aspect-oriented programs (Zhao, 2003). For each aspect or class, the approach performs three levels of testing: intra-module, inter-module, and intra-aspect/intra-class testing. Definition-Use (DU) pairs are constructed to determine what interactions between aspects and classes must be tested. Zhao and Rinard (2003) 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. Control flow graphs are constructed at both system and module level, and test suites are derived from control flow graphs. To reduce the cost of testing aspects, Zhou et al. (2004) have introduced an algorithm based on control flow analysis for selecting relevant test cases. The algorithm evaluates test coverage and selects relevant test cases when existing test cases cannot satisfactorily cover the aspects under test. Xie et al. (2005, 2006) proposed a framework for automatically generating test inputs for AspectJ programs, where a wrapper class is created for each base class under test. The above work focuses on code-based testing. It addresses the question of “how much is the program being covered by testing?” other than “does the program meet the requirements?”. In comparison, this paper focuses on testing conformance between aspect-oriented programs and aspect-oriented design models.

Xu et al. have developed an approach to testing aspect-oriented programs against aspect-oriented state models (Xu et al., 2005; Xu & Xu, 2006a, 2006b). Classes and aspects are all specified by state models. The round trip testing strategy for object-oriented programs (Binder, 2000) is adapted to test generation (Xu et al., 2005). In general, state models are suitable for modeling and testing of individual classes/aspects or small clusters of classes/aspects. They rely on rigorous specification of object states. Different from the state-based approach, this paper exploits aspect-oriented UML models (using class diagrams and sequence diagrams) to derive tests for aspect-oriented programs. These models are more appropriate for capturing the dynamic interactions between aspects and classes. The preliminary idea of this work originates from the position paper presented at the first workshop on testing aspect-oriented programs (Xu and Xu, 2005. It is neither peer-reviewed nor officially published). This paper improves the previous work from the perspectives of aspect-oriented UML models, model-based test generation, and test execution.

AOP can provide a convenient way to develop testing tools or built-in tests (Bruel et al., 2003; Rajan and Sullivan, 2005). The work along this line is not closely relevant to the paper, though. In addition, there are other
approaches to quality assurance of aspect-oriented software. For example, model-checking can be used to verify whether or not an aspect-oriented program satisfies expected properties, such as deadlock, liveness, and fairness (Ubayashi and Tamai, 2002; Denaro and Monga, 2002), or to reason about interactions as the result of weaving (Li et al., 2002). A method for static analysis of aspects was presented based on a syntactic model of pointcut designators (Sereni and Moor, 2003).

2.2 Model-Based Testing

Model-based testing makes use of explicit models of the system under test’s (SUT) intended behavior for generating test cases and verifying conformance between the SUT and its models (Pretschner et al., 2005; El-Far and Whittaker, 2001). A behavior model is an abstract, simplified description of the SUT’s intended behavior. Traces of the model are selected to constitute test cases for the SUT. The key aspects of model-based testing include behavior model, test generation algorithm, and supporting infrastructure for the test execution (e.g., test harness). Model-based testing is appealing because of several benefits (Pretschner et al., 2005; El-Far and Whittaker, 2001), including: (1) the modeling activity helps clarify requirements and enhances communication between developers and testers, (2) design models, if available, can be reused for testing purposes, (3) the model-based testing process can be (partially) automated, and (4) model-based testing can help improve fault detection capability and reduce testing cost by automatically generating and executing many test cases. Pretschner et al. (2004) demonstrated that, for the case study of an automotive network controller, a six-fold increase in the number of model-based tests has led to an 11% increase in detected faults. Dalah et al. (1999) reported an empirical study on four large-scale applications, in which model-based test generation revealed numerous defects that were not exposed by traditional approaches. Using model-based testing methods and tools, Blackburn et al. (2002) were able to identify the software error of the Mars Polar Lander (MPL) that is believed to have caused the MPL to crash to the Mars surface on December 3, 1999.

Modeling is the basis of model-based testing. The modeling languages for testing purposes have ranged from finite state machines (Pretschner et al., 2005; Binder, 2000) and UML statecharts (Offutt et al., 2003) to grammars (El-Far and Whittaker, 2001). Offutt et al. have also explored test generation from UML sequence diagrams (Offutt and Abdurazik, 2002; Abdurazik et al., 2004). Their experimental results suggest that test cases generated from sequence diagrams do better at revealing integration level faults than those generated from statecharts, whereas the latter do better at revealing unit level faults. Andrews et al. have developed a technique for testing executable forms of UML models specified by class diagrams and collaboration diagrams (Andrews et al., 2003). They defined a family of test adequacy criteria for class diagrams (associate-end multiplicity, generalization, and class attribute criteria) and collaboration diagrams (condition coverage, full predicate
coverage, all-message paths, and collection coverage criteria). They have further adapted the technique to sequence diagrams (Pilskalns et al., 2003). None of the above work is concerned with aspect-orientation.

2.3 Aspect-Oriented Modeling

With the development of AOP applications, there is an increasing need for addressing crosscutting concerns in early phases of software development. Aspect-oriented modeling (AOM) is therefore of great interest. AOM involves identifying, analyzing, managing, and representing crosscutting concerns. It targets a simplified, abstract description of an aspect-oriented design. An aspect-oriented modeling method requires three types of constructs for modeling base elements, crosscutting elements, and crosscutting relationships. UML, as the de facto standard for object-oriented modeling, has been a dominant language for specifying base elements of an aspect-oriented model. Recently, extensions to UML have been investigated for modeling such crosscutting elements and relationships as join points, pointcuts, advice, aspects, inter-type declarations, and role-bindings (Aldawud et al., 2002; Aldawud and Bader, 2003; Chavez and Lucena, 2002; Han et al., 2004; Mellor, 2003; Stein et al., 2002; Zavaleta et al., 2004; Elrad et al., 2005; Reddy et al., 2005). In addition, use cases and aspects can be used together for aspect-oriented requirements elicitation and specification (Jacobson, 2003; Jacobson and Ng, 2005; Pawlak and Younessi, 2004).

Modeling, however, is a broad notion that can be involved in various perspectives of software development, such as design specification, code generation, testing, and reverse engineering. Models from different perspectives require different levels of detail although their structures may appear to be similar (Prenninger and Pretscher, 2005). For example, a traditional state model for design specification does not carry sufficient information for test generation. The testable FREE state model results from enhancing a traditional state model with regular expressions (Binder, 2000). The existing aspect-oriented extensions to state models (Mahoney and Elrad, 2005; Aldawud et al., 2002) and UML (Aldawud et al., 2002; Aldawud and Bader, 2003; Chavez and Lucena, 2002; Han et al., 2004; Mellor, 2003; Stein et al., 2002; Zavaleta et al., 2004; Elrad et al., 2005; Reddy et al., 2005) are primarily for the purposes of design specification. Groher and Schulze (2003) have investigated AOM for code generation. For program understanding, Coelho and Murphy (2005) have developed a tool for presenting crosscutting structures in AspectJ programs. Different from the above work, our purposes of aspect-oriented modeling are for testing aspect-oriented programs. We exploit sequence diagrams to model interactions between aspects and objects so as to reveal aspect-related faults.

3. Overview of Our Approach

The testing process of our approach is shown in Figure 1. It consists of the following steps:

1. Building an aspect-oriented UML model according to the design of the SUT,
(2) Generating test cases from the aspect-oriented model,
(3) Executing tests by feeding test inputs to the aspect-oriented program and the test harness, and
(4) Determining whether a test case passes or fails by comparing the actual result of the test execution with
the expected result of that test.

Although the above testing process is similar to that of a traditional model-based testing approach (El-Far and
Whittaker, 2001), the main differences are (1) the models are aspect-oriented, which specify primary concerns
(classes) as well as crosscutting concerns (aspects), and (2) tests are generated from aspect-oriented models. In
other words, we address two issues that are essential to model-based testing of aspect-oriented programs:

- How to model aspect-oriented software in a way that facilitates testing?
- How to generate test cases from aspect-oriented models for detecting aspect-specific faults?

Figure 1. Model-based testing process

An explicit fault model is also critical to an effective testing method. Models for test generation should
facilitate revealing the target faults. The fault model of our approach builds upon the likely faults that are related
to crosscutting concerns in aspect-oriented software, e.g., incorrect strength in pointcut patterns, incorrect aspect
precedence (Alexander et al., 2004), and incorrect advice type. These faults are essentially caused by improper
use of aspect-related programming constructs. To facilitate revealing the faults through model-based testing, we
extend UML to model structure, behavior, and constraints of crosscutting concerns in aspect-oriented software.
Since the UML consists of a great variety of diagrams for specifying a system from various views, this paper
focuses on the modeling constructs for the purposes of test generation, including class diagrams and sequence diagrams, together with the extensions for aspects-related modeling (i.e., aspect diagrams). Class/aspect diagrams specify class and aspect structures and their relationships. As an extension to UML class diagrams, a class/aspect diagram consists of a collection of declarative model elements, such as classes, aspects, types, and their contents. Sequence diagrams are exploited to model dynamic interactions between objects and aspects. We assure the quality of aspect-oriented models by checking the models against the requirements via walkthroughs. We will elaborate on aspect-oriented UML models in Section 4.

Faulty aspects often affect message flows in an unexpected way. Our approach aims to derive message sequences from aspect-oriented models for exercising interactions between classes and aspects in aspect-oriented programs. We specify an aspect-oriented model by class diagrams, aspect diagrams, and sequence diagrams (for methods in classes and advice in aspects). Dynamic system behaviors are determined by weaving sequence diagrams of advice into sequence diagrams of methods. As will be described later, we construct an Aspect-Object Flow (AOF) tree according to a woven sequence diagram and class/aspect diagrams.

4. Aspect-Oriented Modeling

To support model-based testing of aspect-oriented programs, we exploit UML 2.0 to specify the primary and crosscutting concerns in an aspect-oriented design. In our approach, an aspect-oriented model consists of class/aspect diagrams, together with sequence diagrams for class methods and aspect advice. Class/aspect diagrams describe structural properties of primary and crosscutting concerns, respectively. Sequence diagrams with OCL constraints describe dynamic behaviors of primary and crosscutting concerns. OCL constraints represent conditions associated with messages between objects. Based on the UML 2.0 meta-model (Barra et al., 2004), we express crosscutting concerns as in Figure 2, where each aspect has three elements: pointcut, advice, and introduction. Port is a structural feature of a classifier that specifies a distinct interaction point between the 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. In our approach, declarations of pointcut, advice, and introduction in aspect diagrams follow AspectJ (Kiczales et al., 2001).

Figure 3 shows a simple class/aspect diagram with two classes: Purchase and Card from the case study in Section 6. Each class has a join point collected by pointcut expression priceMonitor in aspect AccessControl, which facilitates communication between Purchase and Card objects. Sequence diagrams for class methods and aspect advice are used to model message flows or interactions among objects and aspects. Unlike existing work (Stein et al., 2002) which visualizes pointcut patterns and reflects the weaving mechanism, our approach does not introduce additional crosscutting notations into sequence diagrams as the class/aspect diagrams describe
crosscutting concerns. For example, Figure 4 is a sequence diagram picturing the behavior of `getCharge` and describing how charges are calculated when customers purchase various cards. The sequence diagram consists of three objects, `Purchase`, `Card`, and `Price`, and six messages, `getPrice`, `getUnitPrice`, `getQuantity`, `getDiscountUnitPrice`, `chargeWithinQuantity`, and `chargeBeyondQuantity`. The objects are represented as rectangles with the underlined class name in the rectangle, and the interactions (messages) between different objects in the sequence diagram are denoted by directed arrows. Multiple messages are allowed to be associated with the same directed arrow. The `priceMonitor` pointcut is associated with an `around` advice, which is modeled by the sequence diagram in Figure 5. The advice checks if the caller is authorized. If so, the caller may proceed to invoke `getPrice`; otherwise the access is denied.

![Figure 2. Class/aspect diagrams](image)

![Figure 3. The class/aspect diagram for card purchasing](image)
Figure 4. The sequence diagram for the \textit{getCharge} scenario

Figure 5. The sequence diagram for the advice on \textit{priceMonitor}

Since the dynamic behavior of an aspect-oriented model is determined collectively by aspects and classes, the weaving mechanism for the aspect-oriented modeling is critical to test generation. Before generating tests for a method under test, we compose the method with its advice (i.e. weave their sequence diagrams). The following depicts the general weaving process:

- Finding join points and corresponding aspects in the method sequence diagram based on the class/aspect diagrams,
- Obtaining the sequence diagram of the advice on each join point, and
- Integrating the advice sequence diagram at the join point into the method sequence diagram according to the advice type (before, after, or around).
Figure 6 shows the woven sequence diagram for `getCharge` with the security concern `AccessControl`. The `around` advice on `priceMonitor` in the `AccessControl` aspect suspends the current process and takes over the control. It results in two situations: either stopping the current process or resuming the suspended process (giving the control back).

```
authorized:= authentication(): boolean
```

```
priceMonitor
```

```
AccessControl:
authorized:=
authentication(): boolean
```

```
Login:
Card:
Price:
```

```
[authorized]
```

```
[!authorized]
```

```
unitPrice:= getUnitPrice(): double
discountQuantity:= getQuantity(): double
discountUnitPrice:= getDiscountUnitPrice(): double
```

```
[quantity<discountQuantity]
charge:=chargeWithinQuantity(unitPrice, quantity): double
```

```
[quantity>=discountQuantity]
charge:=chargeBeyondQuantity(discountUnitPrice, quantity): double
```

```
SD getCharge
```

```
5. Test Generation and Execution
```

We aim to generate tests for exercising the interactions between classes and aspects. The expected interactions are implied by woven sequence diagrams obtained from the sequence diagrams and class/aspect diagrams. The general process of generating tests from a woven sequence diagram is as follows:

1. Converting the woven sequence diagram into an AOF tree according to the class/aspect diagrams and coverage criteria for sequence diagrams. In the AOF tree, each path from the root to a leaf node indicates a test template, i.e., an expected sequence of messages.

2. Collecting for each test template the constraints involved in the path. The constraints typically include the corresponding conditions for conditional behaviors in the woven sequence diagram.

3. Assigning values or objects to the input variables in the test template that satisfy the constraints. This results in concrete test cases.
5.1 Construction of AOF Trees

For a woven sequence diagram, we first construct the AOF graph and then convert it into a tree. The AOF graph construction focuses on the objects/aspects and messages that directly interact with the method under test or the advice on the method. Suppose `getCharge` in class `Purchase` in Figure 3 is the method under test. It directly interacts with `Price` objects, for example, through a `getDiscountUnitPrice` message to a `Price` object. Hence, `Price` will be used for the construction of `getCharge`’s AOF graph. Messages within `getDiscountUnitPrice`, if any, will not be considered because their interaction with `getCharge` is indirect. The rationale is `getDiscountUnitPrice` will be tested separately by using the sequence diagram that involves the details of `getDiscountUnitPrice`. In addition, the graph construction applies coverage criteria for sequence diagrams. Although there is a variety of coverage criteria that can be applied to sequence diagrams (Pilskalns et al., 2003; Abdurazik et al., 2004), we focus on condition coverage, polymorphic coverage, and loop coverage as they are important for revealing aspect faults. For example, condition coverage is necessary because each branch of a conditional behavior may be affected by aspects. Polymorphic coverage aims to exercise each potential object type that may be affected by aspects. Suppose an abstract class `C` has three subclasses. When an abstract method `m` in `C` has been picked out by a pointcut, interactions between the aspect and each concrete method `m` in `C`’s subclasses need to be tested. As execution of loops may also determine whether or not aspects are triggered, our approach also takes loop coverage into consideration. Loop coverage for traditional programs has been well-studied (Marick, 1995). For simplicity, we only consider the 0/1/maximum loop coverage and derive test cases to make a loop body executed for 0, 1 or maximum time(s). Note that the 0/1/maximum coverage is not necessarily feasible because, for example, a loop condition may always evaluate true (or false) or the loop body may be executed for a fixed number of times.

An AOF graph is a directed acyclic graph. It is defined as a pair \((V, E)\), where \(V\) is a set of vertices and \(E\) is a set of edges (arcs) between the vertices. We have \(E = \{(u, v) \mid u, v \in V\}\). Each vertex includes four fields: two indices (level number and node number within the same level), class or aspect identity, messages, and constraints. Given the woven sequence diagram for a method under test, we first build a vertex for each behaviour block that directly interacts with the method or the advice, and assign each vertex the corresponding level index, class/aspect identity, and messages (or behaviours). A behaviour block is a sequence of adjacent messages to the same object, a conditional structure, or a repetition structure. The initial constraint for each vertex is empty. Next we connect each of two adjacent vertices with an arc according to their sequential relationship in the woven sequence diagram. Then, for each vertex, we do the following:
(1) If the behaviour is conditional, we replace the vertex with two vertices, labelled with node numbers 1 (the condition is true) and 2 (the condition is false), respectively. This achieves branch coverage for conditional behaviours.

(2) If the behaviour affected by aspects is polymorphic, we expand the vertex at the same level according to the subclasses of the class identity in the vertex. The subclasses are obtained from the class/aspect diagrams. A new vertex is created for each subclass. The node numbers for these vertices start from 1. This achieves polymorphic coverage. If the class involved is an abstract class without implementation (a.k.a. interface), the original vertex can be removed.

(3) If the behaviour is a repetition, we expand the vertex similarly for a conditional behaviour. For simplicity, we only consider the 0/1/maximum coverage for loop bodies, i.e., we make the loop body be exercised for zero, one or maximum time(s) if feasible.

Figure 7. The AOF graph for `getCharge`

An expanded vertex will be connected to the vertex at the next level unless it contains a termination instruction (e.g., ‘X’ in the upper ‘Alt’ block in Figure 6). Consider `getCharge` in Figure 6. Its AOF graph is shown in Figure 7. The graph starts from the top in the sequence diagram of `getCharge`. Two vertices in level 1
represent two possible situations, authorized and unauthorized, of security checking. The vertex representing the authorized situation does not have a successor vertex. The three vertices in level 3 imply three subclasses for the abstract class *Price*, which achieves the polymorphic coverage. Here *Price* is an interface per se. Similarly, the two vertices in level 4 represent the two situations, where \( \text{quantity} \leq \text{discountQuantity} \) and \( \text{quantity} > \text{discountQuantity} \). They achieve branch coverage for the conditional behavior.

Before generating tests, we further expand an AOF graph to an AOF tree. Since AOF graphs are directed and acyclic, the transformation is done by traversing the AOF graph with the depth-first search strategy. In the AOF tree, each path from the root vertex to a leaf vertex is a test template. A concrete test is obtained by creating input data or objects that satisfy the constraints in the template. Figure 8 shows the AOF tree generated from the AOF graph in Figure 7.

![AOF Tree for getCharge](image)

**Figure 8. The AOF tree for getCharge**

### 5.2 Test Generation

Now we discuss how to derive concrete test cases for a given test template in the AOF tree. Since constraints of different vertices in a given path may interact with each other, we must consider the constraints collectively when generating a specific value or object for some variable. For example, a constraint under which an object is invoked at time \( t_1 \) is \( x > 0 \), and later on, another constraint \( x \leq 10 \) is attached to the same object at a different time (say \( t_2 \)). Without considering all the constraints together, a test generation procedure may choose a value (such as \( x = 100 \)) that satisfies the first constraint but not the second. To overcome this problem, we need to generate a set of assignments without violating any constraints (e.g., assigning 5 instead of 100 to \( x \) in the above example). This set of assignments represents the states of an initial message in the test template.

Many existing constraint solvers deal only with numeric data types such as integer, float, boolean, and their arrays (Sy and Deville, 2001; Sy and Deville, 2003; Zeng et al., 2001). Other studies (Gupta et al., 1998; Li et al., 2005; Offutt et al., 1999) made some improvements in the handling of loops and variables of string types, regular
expressions, and objects. However, solving any combination of multiple constraints is in essence an undecidable problem (Binder, 2000). As a result, for the empirical study reported in Section 6 we adopted a procedure to manually resolve the constraints. We collect and consider constraints before choosing suitable test inputs, and select boundary (or random) values for constrained (or unconstrained) variables that satisfy all collected constraints. After an initial assignment of a value (or an object) to a variable, we may need to do a backtracking and a reselection of another appropriate value/object for the same variable if violations with respect to other constraints are identified later on because of the current assignment to the variable (e.g., reassigning 5 to \( x \) to replace the initial assignment of 100 in the above example). Nevertheless, our experience suggests that the constraint solving procedure we adopted served our purpose for deriving tests that trigger the execution of aspects. Of course, we also realize that the procedure we have used may not be efficient when multiple constraints are involved.

As an example, let us consider the AOF path 1.1-2.1-3.1-4.1 in Figure 8 for test generation. For discussion purposes, we list the nodes in Table 1; each entry includes the class/aspect, the behavior of the object/aspect, the corresponding constraint, and variables/objects to be resolved and their currently assigned values. Our goal is to derive a test, i.e., a \texttt{getCharge} message to a \texttt{Purchase} object, so that \texttt{AccessControl}, \texttt{Card}, \texttt{HolidayPrice}, and \texttt{this} (the class of \texttt{getCharge}, i.e., \texttt{Purchase}) are exercised in a specified order. Using backward-chaining reasoning, we start with the last entry (Node 4 in Table 1) and try to satisfy the constraint “\texttt{quantity} < \texttt{discountQuantity}.” Since \texttt{discountQuantity} is an intermediate variable, it is computed in the previous steps. Therefore, \texttt{quantity} is put into the last column as the variable to be resolved. We try to assign a random number (say 1) to \texttt{quantity} and then move on to the entry at Node 3 in Table 1. As node 3 does not have a new constraint, we only retain the previous constraint “1<\texttt{discountQuantity}.” In this step, we need to resolve variables \texttt{unitPrice}, \texttt{discountQuantity}, and \texttt{discountUnitPrice}. As there is no constraint on \texttt{unitPrice} or \texttt{discountUnitPrice}, we assign random values 3.0 and 2.0 to them, respectively. For \texttt{discountQuantity}, it must be greater than 1. We assign it the boundary value 2. These indicate what is needed for the \texttt{HolidayPrice} object. Now we move on to the entry at Node 2 in Table 1: the \texttt{getPrice} message to the \texttt{Card} object returns the \texttt{HolidayPrice} object used at Node 3. Similarly, for Node 1 in Table 1, the test inputs must make \texttt{authentication()} true. Now the remaining issue is how to create objects (of \texttt{HolidayPrice}, \texttt{Card}, and \texttt{Purchase}) for the test to exercise the \texttt{getCharge} method of \texttt{Purchase}. To do so, we examine the diagrams of the classes involved; when class diagrams do not provide such detailed information about constructor parameters, we use class interfaces (signatures of public constructors and methods). For \texttt{HolidayPrice}, its class interface declares a constructor with three parameters (\texttt{unitPrice}, \texttt{discountQuantity}, \texttt{discountUnitPrice}), and thus we have the information to create the \texttt{HolidayPrice} object: \texttt{HolidayPrice hp = new HolidayPrice(3.0, 2, 2.0)}. For \texttt{Card}, its constructor has two parameters, a \texttt{string} object and a \texttt{Price} object. Thus we can use \texttt{Card c = new Card(hp, “random”)} to create a \texttt{Card} object (where ‘random’ refers to a random string
object). For Purchase, its constructor is of the form: Purchase (Card c, int quantity). We use the Card object c and quantity (which is an integer with a value 1) to create the Purchase object: Purchase p = new Purchase(c, quantity). Putting the above together forms the following concrete test:

```java
HolidayPrice hp = new HolidayPrice(3.0, 2, 2.0);
Card c = new Card(hp, "random")
Purchase p = new Purchase(c, 1);
p.getCharge();
```

A correctly implemented getCharge method of Purchase, when exercised by the above test, should behave as defined in Table 1 (i.e., follow the AOF path 1.1-2.1-3.1-4.1 in Figure 8). Of course, as the AccessControl aspect is automatically weaved and executed, the logon input must make the authentication true.

### Table 1. A sample test template for test generation

<table>
<thead>
<tr>
<th>No</th>
<th>Class/Aspect</th>
<th>Method/Advice</th>
<th>Constraint</th>
<th>Variables/objects to be resolved and their currently assigned values</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AccessControl</td>
<td>authorized:= authentication(): boolean</td>
<td>authorized = true</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Card</td>
<td>price:= getPrice (): HolidayPrice</td>
<td></td>
<td>Card (HolidayPrice)</td>
</tr>
<tr>
<td>3</td>
<td>HolidayPrice</td>
<td>unitPrice:=getUnitPrice(): double</td>
<td></td>
<td>HolidayPrice (unitPrice = 3.0 discountUnitPrice = 2.0 discountQuantity = 2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>discountQuantity:=getQuantity(): double</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>discountUnitPrice:=getDiscountUnitPrice()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>this (Purchase)</td>
<td>charge:=chargeWithinQuantity(unitPrice, quantity): double</td>
<td>quantity&lt;= discountQuantity</td>
<td>Quantity = 1</td>
</tr>
</tbody>
</table>

### 5.3 Test Harness

Once objects are created for a concrete test, we can execute the aspect-oriented system against this test case. To validate the execution result, we need to check whether or not the execution produces the same result as expected. Traditionally, the expected result or oracle value of a test case is defined in terms of the computational outcome. Nevertheless, defining the exact oracle value of each test is either difficult or tedious (Binder, 2000). Recall that our objective is to verify whether or not an aspect has implemented the crosscutting feature correctly. To do so, we need to (1) define our expected crosscutting feature and (2) keep track of the actual crosscutting implementation so as to compare it with the expected feature. The expected feature is actually indicated in the test template (i.e., message sequence). For (2), we exploit the AOP paradigm to realize a test harness. It keeps track of the actual message sequences and compares them with the expected message sequences.

The aspect-oriented test harness monitors objects and aspects involved in the test execution by automatically tracing the method calls, method executions, and advice executions. This is done by a generic (i.e., abstract) trace aspect that contains three pointcuts: traceBase, tracePolymorphism, and traceCrosscut. The traceBase pointcut
monitors objects that directly interact with the object under test, the *tracePolymorphism* pointcut monitors the function executions of polymorphic objects, and the *traceCrosscut* pointcut monitors all the executions of advice. Consider `getCharge` in our example; the *traceBase* pointcut collects all the method calls `getCharge` made. All objects (*Card, Price*) and aspects (*AccessControl*) which directly interact with `getCharge` are monitored; the *tracePolymorphism* pointcut collects method executions in the subclasses of *Price*, i.e., *HolidayPrice, BirthdayPrice, and ValentinePrice*; the *traceCrosscut* pointcut collects all the advice in the *AccessControl* aspect. Since the pointcut definitions depend on concrete test cases, the test harness automatically generates a trace aspect for each test according to the given methods and classes in the test.

6. An Empirical Study

In this section, we first introduce the subject application, present the experimental results, and then discuss the threats to validity and limitations.

6.1 The Application

We have applied the above approach to the testing of the greeting card purchase subsystem of an online shopping application. The subsystem consists of 12 classes, 3 aspects and 398 lines of code. The application supports flexible business rules for a variety of greeting cards. The price of a card in a purchase depends on the card type and the quantity of the cards. The cards are categorized as holiday cards, birthday cards, valentine cards, etc. Different types of cards have different unit price, minimum quantity of cards for discount, and discounted unit price. Discounted unit price is available only when the quantity is greater than the required minimum. The following base classes address the core functional concerns:

- **Card**: a card has information such as code and price.
- **Price**: an abstract class. The price of a specific type of card depends on the business rule.
- **HolidayCardPrice**: includes unit price, minimum quantity for discount, and discounted price for holiday cards.
- **BirthdayCardPrice**: includes unit price, minimum quantity for discount, and discounted price for birthday cards.
- **ValentineCardPrice**: includes unit price, minimum quantity for discount, and discounted price for valentine cards.
- **Purchase**: a purchase includes a particular type of card and the quantity of the cards.
- **Transaction**: a transaction consists of a number of purchases.

Multiple price classes are designed and implemented in order to avoid potential code smells when the pricing policies for different cards become more complex. Code smells are “characteristics of code (e.g., making a single
change causes several other changes) that indicate less than acceptable quality” (Fowler, 1999) and thus often need to be addressed by refactoring and even redesigning. Protection of sensitive data and handling of exceptions and errors are the two separate concerns regarding security and usability. They are addressed by the following aspects:

- **AccessControl**: This aspect is responsible for distinguishing legal and illegal access to sensitive data, such as price information. Each type of card has its own price information. The base class *Card* may expose the price information to any request. For experimental purposes, we use a password-based access control for security check.

- **NullPointerExceptionHandling**: This aspect catches and handles *NullPointerException* exceptions consistently at run time. It will avoid inconsistent exception handling, i.e., some method calls have exceptions while others do not.

*Figure 9. Classes and Aspects of the Card Purchase Application*
Figure 10. The sequence diagram for the exceptionMonitor advice

Figure 9 shows how the two aspects crosscut the base classes. The aspects are indicated by dashed circles. The details of the aspect AccessControl is already shown in Figure 3. Figure 10 is the advice sequence diagram for the exceptionMonitor pointcut of the NullPointerExceptionHandling aspect, which picks out join points of ports 3 and 4 in Purchase and Transaction, respectively. Figure 11 shows part of the woven sequence diagram for the getTotalCharge method in the Transaction class where the exceptionMonitor advice applies to several messages with a potential null pointer.

Figure 11. Part of the woven sequence diagram for the getTotalCharge method in the Transaction class

6.2 Experimental Results
Our experiment targets four types of aspect-specific faults, including incorrect advice type, weak or strong pointcut strength, and incorrect aspect precedence. A fault of an incorrect advice type refers to using a type of advice different from the one defined in the design (for instance, incorrectly using `after` instead of `before`). A weak (or strong) pointcut means the implementation picks out an unnecessary (or misses a specified) join point. Based on these fault types, we use four mutant operators to guide the seeding of faults into the working code of aspects. The mutant operators include pointcut strengthening (PCS), pointcut weakening (PCW), precedence changing (PRC), and advice type changing (ATC). PCS, PCW, and PRC come from (Mortensen and Alexander, 2005), which produce incorrect strong pointcuts, incorrect weak pointcuts, and incorrect aspect precedence, respectively. We also introduce a new mutant operator ATC for seeding faults of incorrect advice types. Using these mutant operators, we create 7 faulty programs (two by PCS, PCW, and ATC, respectively, and one by PRC). PCS, PCW, ATC are applied to `AccessControl` and `NullPointerExceptionHandling`. PRC is used to create a fault with incorrect precedence between the two aspects. The test suite consists of 13 test cases derived from the AOF trees of the woven sequence diagrams in Figure 6 and Figure 11 according to the condition, polymorphic and loop coverage criteria. These tests are applied to all the faulty programs.

Table 2. Sample test cases for revealing different types of faults

<table>
<thead>
<tr>
<th>Type of Fault</th>
<th>Model</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incorrect advice type (ATC)</td>
<td><code>AccessControl</code></td>
<td><code>After/withincode(double Purchase.getCharge()&amp; &amp; call(* Card.get*()))/NA</code></td>
</tr>
<tr>
<td>Weak pointcut (PCW)</td>
<td><code>AccessControl</code></td>
<td><code>Around/withincode(double Purchase.getCharge()&amp; &amp; call(* Card.get*()))/NA</code></td>
</tr>
<tr>
<td>Strong pointcut (PCS)</td>
<td><code>AccessControl</code></td>
<td><code>Around/withincode(double Purchase.getCharge()&amp; &amp; call(* * get*()))/NA</code></td>
</tr>
<tr>
<td>Incorrect precedence (PRC)</td>
<td><code>AccessControl</code>, <code>NullPointerException</code></td>
<td><code>NA/NA/AccessControl, NullPointerException</code></td>
</tr>
</tbody>
</table>

Table 2. Sample test cases for revealing different types of faults

<table>
<thead>
<tr>
<th>Type of Fault</th>
<th>Model</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incorrect advice type (ATC)</td>
<td><code>AccessControl</code></td>
<td><code>After/withincode(double Purchase.getCharge()&amp; &amp; call(* Card.get*()))/NA</code></td>
</tr>
<tr>
<td>Weak pointcut (PCW)</td>
<td><code>AccessControl</code></td>
<td><code>Around/withincode(double Purchase.getCharge()&amp; &amp; call(* Card.get*()))/NA</code></td>
</tr>
<tr>
<td>Strong pointcut (PCS)</td>
<td><code>AccessControl</code></td>
<td><code>Around/withincode(double Purchase.getCharge()&amp; &amp; call(* * get*()))/NA</code></td>
</tr>
<tr>
<td>Incorrect precedence (PRC)</td>
<td><code>AccessControl</code>, <code>NullPointerException</code></td>
<td><code>NA/NA/AccessControl, NullPointerException</code></td>
</tr>
</tbody>
</table>

Table 2. Sample test cases for revealing different types of faults

<table>
<thead>
<tr>
<th>Type of Fault</th>
<th>Model</th>
<th>Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Incorrect advice type (ATC)</td>
<td><code>AccessControl</code></td>
<td><code>After/withincode(double Purchase.getCharge()&amp; &amp; call(* Card.get*()))/NA</code></td>
</tr>
<tr>
<td>Weak pointcut (PCW)</td>
<td><code>AccessControl</code></td>
<td><code>Around/withincode(double Purchase.getCharge()&amp; &amp; call(* Card.get*()))/NA</code></td>
</tr>
<tr>
<td>Strong pointcut (PCS)</td>
<td><code>AccessControl</code></td>
<td><code>Around/withincode(double Purchase.getCharge()&amp; &amp; call(* * get*()))/NA</code></td>
</tr>
<tr>
<td>Incorrect precedence (PRC)</td>
<td><code>AccessControl</code>, <code>NullPointerException</code></td>
<td><code>NA/NA/AccessControl, NullPointerException</code></td>
</tr>
</tbody>
</table>
Handling | Purchase | AccessControl | Purchase
---|---|---|---

Table 2 shows an example for each of the target fault types. Each row is for a specific fault type. It includes (1) the specification of advice type, pointcut, and aspect precedence, (2) expected message sequence derived from the AOF paths, (3) the actual implementation of advice type, pointcut, and aspect precedence, and (4) the actual AOF sequence collected by the test harness. Let us take the first row as an example. The expected AOF sequence (AccessControl→Card→Price→HolidayPrice→Purchase) is different from the actual AOF sequence (Price→AccessControl→HolidayPrice→Purchase). The difference in the AOF sequences helps us reveal the difference between the specification and implementation and detect the corresponding fault: the advice type is changed from around (in the specification) to after (in the implementation). The second row describes a weak pointcut fault, where call(* Card.get*) is replaced with call(* *.get*). The specification only picks out calls to the get method of Card; the implementation, however, picks out calls to get of any class. The third row describes a strong pointcut fault where call(* *.get*()) is replaced with call(* Card.get*()). For both cases, we observe a difference between the expected and the actual AOF sequences and use such information to detect the corresponding faults. The fault described in the fourth row belongs to the type of incorrect aspect precedence between the two aspects AccessControl and NullPointerException. The implementation uses a wrong precedence. Once again, we can detect the corresponding fault by examining the difference between the expected and the actual AOF sequences.

In short, we have demonstrated that our approach can be effective in detecting the faults described by our fault models including incorrect advice type, incorrect pointcut strengths (weaker or stronger), and incorrect aspect precedence.

6.3 Discussion

It is known that controlled experiments have potential threats to validity (Rothermel and Harrold, 1998). Our empirical study is no exception. The threats to validity in our experiments fall into three categories: (1) threats to construct validity. A main issue is whether other effects could be responsible for the results, e.g., fault detection ability. This research has targeted four common types of aspect faults. To improve the results, our future work will include more fault types of AOP. (2) Threats to internal validity. A main issue is whether the cost-benefit factors are appropriate. In our experiments, these factors include the modeling cost, the number of test cases, and fault-detection ability. Our future work will consider other factors, e.g., time for deriving and executing the test cases. (3) threats to external validity. They are about how representative the subjects are and to what extent the results generalize. Our test generation approach is general – it works as long as the design can be specified by the
aspect-oriented modeling. The aspects in the empirical study, however, do not necessarily reflect all features of aspect-oriented applications. Our future work will apply our approach to large-scale AOP applications.

7. Conclusions

We present an approach to testing whether or not an aspect-oriented program conforms to its expected crosscutting behavior modeled by class/aspects and sequence diagrams in UML. The approach can help testers reveal several types of faults that are specific to aspectual structures, such as incorrect advice type, strong or weak pointcut expressions, and incorrect aspect precedence. Currently, our approach relies on manual derivation of concrete test inputs from the test templates in AOF trees due to the inherent difficulty of the constraint satisfaction problem. Nevertheless, it is of interest to investigate how our approach can be enhanced with limited automation of test generation. For example, we can develop a tool that provides heuristics for test input selection or suggests appropriate alternatives. This will reduce the amount of backtracking, which is often difficult to manage during manual constraint resolution. We can also explore a well-defined, restricted formalism for design modeling to facilitate automated generation of test inputs for some domain-specific aspect-oriented programs.

In addition, an aspect-oriented program may exhibit other types of faults than those discussed in this paper. Primary examples are failure to establish postconditions and failure to preserve state invariants (Alexander et al., 2004). These failures may occur in those aspects that are used to enforce the design contracts of their base classes (Diotalevi, 2004). To reveal such failures, contracts enforced by aspects must be specified rigorously. Our future work will investigate how preconditions, postconditions, and state invariants represented by UML OCL can be used as additional constraints to generate test cases. Our future work will also investigate the correlation between the test coverage criteria for UML models and the fault detection ability of the model-based testing method.

Acknowledgments

This work was supported in part by the ND NASA EPSCoR program through NASA grant #NCC5-582 and by ND EPSCoR IIP-SG program through NSF grant EPS-047679. The authors would like to thank Dr. Linzhang Wang for his input on an earlier draft of this paper.

References


Zhao, J., 2003. Data-flow-based unit testing of aspect-oriented programs, Proc. of COMPSAC’03, pp.188-197, Dallas, Texas, USA.