

State-Based Testing of Integration Aspects

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

Aspect-oriented programming supports a variety of composition strategies, from the clearly acceptable to the questionable. One of the strategies is to make an aspect integrate separate concerns. Such integration aspects, like other aspects, may introduce various programming faults, including incorrect join points, pointcuts, and advice. This paper presents a preliminary study on the state-based testing of integration aspects. We exploit aspect-oriented state models for specifying integration aspects, compose state models of aspects and classes, and generate test cases for integration aspects from their state models. We exercise integration aspects through the interface of their base classes. We also discuss the issues of testing deeply crosscut integration aspects through the clients of their base classes.

Categories and Subject Descriptors

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

General Terms

Verification, Design

Keywords

Aspect-oriented programming, model-based testing, state model, aspect-oriented modeling, integration aspects, integration testing.

1. INTRODUCTION

Aspect-oriented programming (AOP) [8][9] supports a variety of composition strategies, from the clearly acceptable to the questionable [11]. One of the strategies is to make an aspect integrate separate concerns. For instance, in the *Telecom* simulation application of the AspectJ toolkit [1], the *Connection* and *Timer* classes implement two separate concerns as their names suggest. The *Timing* aspect integrates these concerns as a whole by advising *Connection* when to start and stop a *Timer*. Specifically, *Connection* is the base class of the *Timing* aspect, whereas the *Timer* class is integrated with *Connection* through *Timing*. We refer to aspects like *Timing* as *integration aspects*, classes like *Connection* as *base classes*, and classes like *Timer* as *integrated classes*. Typically, an integration aspect composes

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

WTAOP'06, July 2006, Maine, USA.

© 2006 ACM 1-59593-415-4/ 06/ 0007 5.00

multiple classes that implement separate concerns. It can crosscut multiple base classes and involve one or more integrated classes (in contrast, incremental modification aspects in [13] are modeled only with respect to base classes). As base classes and integrated classes can be implemented, executed and unit-tested without integration aspects, testing of integration aspects is in essence a special paradigm of integration testing. The special feature is that integration aspects at the meta-level of classes provide a well-structured way to specify integration of classes.

Integration aspect, like incremental modification aspects [13], may introduce various faults, including incorrect join points, pointcuts, and advice [4]. This paper presents a preliminary study on the state model-based testing of integration aspects. We exploit aspect-oriented state models for specifying classes as well as integration aspects. We compose state models of aspects and classes and generate test cases from their state models. We exercise integration aspects through the interface of their base classes. We also discuss the issues of testing deeply crosscut integration aspects through the clients of their base classes.

The rest of this paper is organized as follows. Section 2 introduces specification of classes and integration aspects. Section 3 describes how to test an integration aspect through the interface of its base classes. Section 4 presents a case study on testing integration aspects. Section 5 discusses the needs and issues of testing integration aspects through the clients of their base classes. Section 6 reviews related work. Section 7 concludes this paper.

2. SPECIFYING INTEGRATION ASPECTS

We use state models to capture both intra-object behaviors of classes and inter-object behaviors of integration aspects. A state model consists of states, events, and transitions. A transition (s_i, e, ϕ, s_j) means that, at state s_i , event e leads to state s_j under guard condition ϕ . As each state is associated with some class, for clarity, we refer to the object state s of class C as $C.s$. Events are related to public constructors and methods of classes. As such, we interpret each transition $(C_i.s_i, e, \phi, C_j.s_j)$ as follows:

- $C_i.s_i$ and $C_j.s_j$ are abstract states of objects of class C_i and C_j , respectively;
- e is corresponding to a method (or constructor), say $m(\tau_1 v_1, \tau_2 v_2, \dots, \tau_k v_k)$, in the specification of class C_j , where τ_i ($1 \leq i \leq k$) is the type of parameter v_i . τ_i can be a fundamental data type or an object type (i.e. class).
- ϕ is a logical condition constructed by using constants, instance fields of class C_j , or explicit parameters v_i ($1 \leq i \leq k$) of method m in class C_j . If τ_i is an object type and f is a public function (method with a return value) of τ_i , then function call $v_i.f$ is allowed to occur in logical formulas.

- $(C_i.s_i, e, \phi, C_j.s_j)$ indicates a call to method (or constructor) m in class C_j under state $C_i.s_i$ that satisfies guard condition ϕ and results in state s_j of C_j object.

For convenience, we use α to represent the dumb state of an object before it is constructed (as in [5]) and new to represent the event for object construction. Hence, transition $(\alpha, new, \phi, C.s_0)$ refers to the construction of an object of class C under condition ϕ (ϕ is optional), which results in the initial state s_0 of the object. Transition $(C_i.s_i, e, , C_j.s_j)$, where ϕ is omitted, means that the transition is unconditional: any event e (or call to method m of class C_j) under state $C_i.s_i$ (typically a message sent by a C_i object) leads to state $C_j.s_j$. We denote the sequence of transitions:

$$(\alpha, new, \phi_0, C_0.s_0), (C_0.s_0, e_1, \phi_1, C_1.s_1), \\ (C_1.s_1, e_2, \phi_2, C_2.s_2), \dots, (C_{n-1}.s_{n-1}, e_n, \phi_n, C_n.s_n)$$

by $\langle new[\phi_0], C_0.s_0, e_1[\phi_1], C_1.s_1, e_2[\phi_2], C_2.s_2, \dots, e_n[\phi_n], C_n.s_n \rangle$. Such a sequence indicates an *abstract test case*. It is abstract in the sense that the parameters of constructor (i.e. new) and method calls are not yet assigned concrete values. A test case starts with a special transition like $(\alpha, new, \phi_0, C_0.s_0)$. It means that, to test a class or its integration with other classes, we have to construct an object of this class. If $C_i=C_j$ for all transitions $(C_i.s_i, e, \phi, C_j.s_j)$ in a state model, the model specifies the intra-object behaviors of a single class C_i . A transition $(C_i.s_i, e, \phi, C_j.s_j)$ where $C_i \neq C_j$ indicates inter-object interaction between two different classes.

Fig. 1 shows the state model of class *Foo*, which has two public methods fe_1 and fe_2 (fe_2 has a formal parameter x). Fig. 2 shows the state model of class *Bar*, which has two public methods be_1 and be_2 . Both *Foo* and *Bar* have a public constructor.

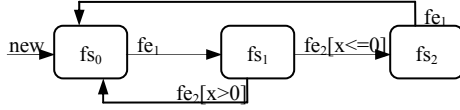


Figure 1. Class *Foo* state model

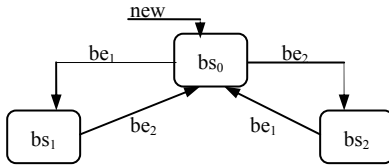


Figure 2. Class *Bar* state model.

Join points are transitions and states in the state models of base classes; a transition/state pointcut picks out a group of join points (transitions/states from one or more state models); a piece of advice is specified as a state model; and an aspect is an encapsulated entity of pointcuts and their advice models. For a transition join point, say $(C.s_i, e, \phi, C.s_j)$, in a pointcut specification, $C.s_i$, ϕ and $C.s_j$ are optional. $(, e, \phi,)$ means that any transition involving e and ϕ in the base model is picked out by the pointcut; whereas $(, e, ,)$ means that any transition involving e is picked out (regardless of the guard condition). In these cases, transition join points are essentially defined with respect to events.

An integration aspect consists of state pointcuts, transition pointcuts, and advice models. An advice model specifies how base classes interact with the integrated classes. It is defined by using pointcut parameters, states and events from the state models

of integrated classes. Fig.3 shows a simple aspect model that integrates the state models of *Foo* and *Bar*. The aspect defines one transition pointcut, i.e. $tcut_1$, which picks out the only transition $(fs_1, fe_2, x \leq 0, fs_2)$ in class *Foo*. In $tcut_1$'s advice model, $Bar.bs_0$ and $Bar.bs_1$ are two states in *Bar*'s state model. Transition $\langle ps_1, new, , Bar.bs_0 \rangle$ involves two classes: *Foo* (picked out by the pointcut) and *Bar*. It means that, under state ps_1 (i.e. $Foo.fs_1$), event new without guard condition is issued to create a *Bar* object with initial state bs_0 . Then the be_1 event indicates a message sent to the *Bar* object. From the coding perspective, such a message is typically sent by the integration aspect. It is somehow different from base class events (messages that will be provided by test cases). Instead, it is similar to actions in a base class state model, which is not part of test input¹. However, events like $Bar.be_1$ are used in the state model of an integration aspect because their resulting states are of interest to integration testing, which involves verification of composite states of multiple objects. Typically, a base class state model does not specify the impact of actions (e.g. sending a message to another object) on the states of *other* objects.

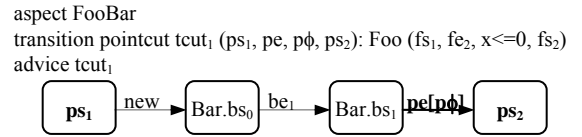


Figure 3. The *FooBar* aspect

Although an advice model involves states and events of integrated classes (e.g. *Bar* in Fig. 3), it does not provide a complete behavior model for the integrated classes. The state model of each class is described separately (e.g. the *Bar* state model in Fig.2). In fact, an advice model specifies how the integrated classes are used together with the base classes. For a transition $\langle s_1, e, \phi, C.s_2 \rangle$ in an advice model, where s_2 is a state in the state model of integrated class C , there must exist at least one state $C.s$ such that $\langle C.s, e, \phi, C.s_2 \rangle$ is a valid transition in C 's state model (e.g. $\langle Bar.bs_0, be_1, , Bar.bs_1 \rangle$ in Fig.3 is also a transition in *Bar*'s state model in Fig.2). A special situation is that $C.s$ is α and e is the new event (e.g. $\langle ps_1, new, , Bar.bs_0 \rangle$ in Fig.3). This type of constraints can be verified automatically against the state models of integrated classes when an advice model is defined or weaved.

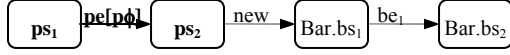
An advice model can simulate the three types (i.e. *before/after/around*) of advice in AOP that imply different styles of integration. The typical situations are as follows:

- In a *before* type advice for pointcut $(ps_1, pe, p\phi, ps_2)$, ps_2 is reachable from ps_1 and $(s, pe, p\phi, ps_2)$ is the last transition to reach ps_2 from some state s . For example, the advice in Fig. 3 is of *before* type, where $s=Bar.bs_1$.
- In an *after* type advice for pointcut $(ps_1, pe, p\phi, ps_2)$, ps_2 is reachable from ps_1 and $(ps_1, pe, p\phi, s)$ is the first transition from state ps_1 to some other state s . Fig. 4(a) shows an example, where $s=ps_2$.

¹ Actions associated with an event in a traditional state model specify the functionality of the method that is corresponding to the event. Typically, they are part of the method implementation and executed when the method is invoked with a valid guard condition. As such, actions are not part of test input.

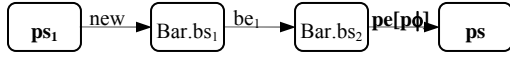
- In an *around* type advice for pointcut ($ps_1, pe, p\phi, ps_2$), ps_2 may or may not be reachable from ps_1 . Fig. 4 (b) shows an example, where ps is the parameter of *state pointcut* $scut_1(ps): Foo.fs_0$. It means that, for a *Foo* object, event fe_1 under the state fs_1 will lead to the state fs_0 , rather than fs_2 .

To maintain a high-level of abstraction, the advice types are implied in advice models, rather than explicitly defined as in AOP. However, the above interpretation can help to detect incorrect advice types in the implementation of integration aspects.



(a) An *after* advice

aspect FooBar2
state pointcut scut (ps): Foo (fs₀)
transition pointcut tcut₁ (ps₁, pe, pφ, ps₂): Foo (fs₁, fe₂, x<=0, fs₂)
advice tcut₁



(b) An *around* advice

Figure 4. Advice examples

The semantics of integration aspects essentially depends on the weaving mechanism that composes aspect models into base class models. The general idea of weaving an aspect model into a base class model is to replace each transition/state join point picked out by some transition/state pointcut in the aspect with the corresponding advice model. Fig. 5 shows the woven model for the *FooBar* aspect in Fig.3. In *Foo*'s state model, the transition ($fs_1, fe_2, x \leq 0, fs_2$) indicates that, under state fs_1 of a *Foo* object, event fe_2 with condition $x \leq 0$ reaches state fs_2 . When *Foo* and *Bar* are integrated by the *FooBar* aspect, under state fs_1 of a *Foo* object, event fe_2 with condition $x \leq 0$ will not reach state fs_2 until a *Bar* object is created and a message be_1 to the *Bar* object is issued (or until states bs_0 and bs_1 , of a *Bar* object have been reached). The initial state of a state model is $C.s_0$ such that $(\alpha, new, \phi, C.s_0)$ and C is the base class. If $(s_1, new, \phi, C'.s_0)$ is a transition but $s_1 \neq \alpha$, then $C'.s_0$ is not viewed as the initial state (in this case C' must not be the base class). In Fig.5, the initial state is fs_0 , not $Bar.bs_0$. Construction of the *Bar* object is typically implemented in the integration aspect.

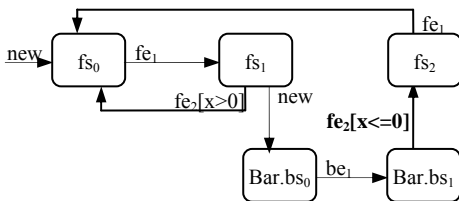


Figure 5. The woven model for the *FooBar* aspect

It is worth pointing out that different implementations may exist for a woven state model. They may or may not use integration aspects. For example, one can simply revise the *Foo* class code by constructing and storing a *Bar* object in an instance field and sending messages to the *Bar* object in *Foo*'s methods. However, mixture of concerns as in this case is exactly a programming problem that AOP was motivated to address.

3. TESTING INTEGRATION ASPECTS

This section introduces the testing process for classes and aspects and describes test generation from state models.

3.1 Testing Process and Automation

Before testing integration aspects, we first test classes. Both are based on the state models. As testing of integration aspects can reuse many results of class testing, we first describe the process of state-based class testing. As shown in Fig. 6, it starts with class modeling, which produces the state models of classes as well as specification of state predicates (Step 1 in Fig. 6). A state predicate defines the relationship between an abstract state used in state models and concrete state (instance) variables in class implementation. For example, *Overdrawn* (i.e. negative balance) is often used as an abstract state in the state model of *BankAccount* class [13]. The implementation of *BankAccount* class, however, typically uses *balance* as the instance field (or state variable). Therefore, we can define state predicate *overdrawn* as $balance < 0$, or $getBalance() < 0$ which bridges the gap between abstract state *Overdrawn* and concrete state variable *balance*. This is crucial to automating the decision making on whether a test case passes or fails by verifying actual object states against expected states. For some classes, abstract and concrete states may be similar. In this case, the value of the state variable is actually one of the abstract states. In the *Telecom* simulation application, the *Timer* class model has two states (*Stopped* and *Started*), whereas the *Timer* class implementation uses an instance variable which value is an enumerated one, either *Stopped* or *Started*. The state predicate for *Stopped* (*Started*) is $getState() == Stopped$ (*Started*). For test automation, a state predicate is simply implemented as a predicate method (i.e. a method with a boolean type of return values).

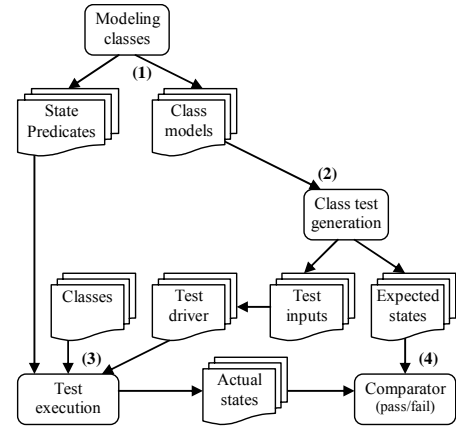


Figure 6. Process of class testing

After the state models of classes are built, we generate abstract tests by converting each state model to a transition tree as in [5] (refer to Step 2 in Fig.6). Each path from the root to a leaf in a transition tree is an abstract test because the constructor and method invocations do not have concrete values for their parameters. Of course, an abstract test reduces to a concrete test case if none of the constructors and methods needs any parameters. Note that, each state model here is for an individual class. Derivation of concrete test cases from an abstract test relies on manual assignment of concrete values to parameters of constructors and methods. This is known as the *path sensitization* problem, which is in general undecidable [5]. Consider function

$y=f(x)$, we want y to be a specific value or in a specific range of values, what is right value or range of values for x ? In order for this problem to be solvable, the function $f(x)$ must be reversible. It is thus mathematically hard (e.g. $f(x)$ could be a complex polynomial).

After concrete tests are specified, we exercise the classes under test (Step 3 in Fig. 6) and compare the actual object states with the expected states to determine whether each test passes or fails (Step 4 in Fig. 6). Except for the class modeling and derivation of concrete test cases from abstract tests, the process of state-based class testing in Fig. 6 is automated.

Now we are ready to discuss the state-based testing process for integration aspects. As shown in Fig.7, it consists of four similar steps. The first step is to build state models for integration aspects. In the second step, we generate tests for integration aspects from base class models, aspect models and concrete base class tests. The last two steps are executing the aspect programs together with the classes and compare the actual object states with expected ones. In summary, testing of integration aspects reuses class models, state predicates as well as concrete base class tests.

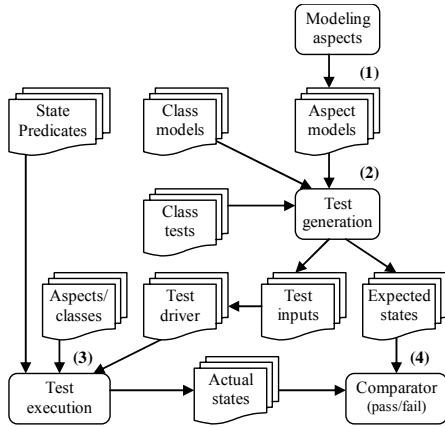


Figure 7. Testing process of integration aspects

3.2 Test Generation

We test integration aspects through testing of their base classes because aspect code is automatically weaved into base classes. Each test is a sequence of the construction of a base class object followed by method invocations to the object. This bears similarity to class testing.

For class testing, we extend the modal class test design pattern for object-oriented programs [5], which derives a test suite by transforming a single class state model to a transition tree and identifying sneak paths with illegal state transitions. Given a state model M , with initial state s_0 , we transform state model M into a transition tree with sneak paths as follows (adapted from [5][13]):

- (1) The root node of the transition tree is s_0 , the initial state of the state model M . We also associate the *new* event and its guard condition (if any) with s_0 .
- (2) For each non-terminal leaf node (say s) in the transition tree.
 - (2.1) For any transition (s, e, ϕ, s') , draw a new edge labeled with $e[\phi]$ and a new node labeled with s' . If s' already appears anywhere in the tree, mark the node as terminal.
 - (2.2) For any event e in the base model that does not transform s to any other state in M , draw a new edge

labeled with e , draw a new node labeled with s (i.e. the state remains unchanged if an illegal event happens. This does not mean that event e transforms state s to state s), and mark the node as terminal and negative. (The path from the root to this node is a sneak one).

- (2.3) For any event e in the base model that does transform s to some other state in M . Suppose $(s, e, \phi_1, s_1), (s, e, \phi_2, s_2), \dots, (s, e, \phi_k, s_k)$ are all the transitions in M that involve e under s . If $(\phi_1 \vee \phi_2 \vee \dots \vee \phi_k)$ does not always evaluate true, then draw a new edge labeled with e [*not* $(\phi_1 \vee \phi_2 \vee \dots \vee \phi_k)$], draw a new node labeled with s , and mark the node as terminal and negative. (The path from the root to this node is a sneak one)

- (3) Repeat (2) until all leaf nodes are terminal;

Figure 8 shows the transition tree for the *Foo* class. Each path from the root to a terminal leaf is an abstract test (it is a negative one if it contains a dashed arrow), which can be used to derive many concrete test cases. For example, the path $\langle new, fs_0, fe_1, fs_1, fe_2[x \leq 0], fs_2, fe_1, fs_0 \rangle$ is an abstract test, whereas $\langle new, fs_0, fe_1, fs_1, fe_2(-1), fs_2, fe_1, fs_0 \rangle$ is a concrete test case with $x=-1$. The test input includes constructor and method invocations together with their parameters (*new, fe₁, fe₂(-1), fe₁*) and the expected result includes the corresponding object states (*fs₀, fs₁, fs₂, fs₀*).

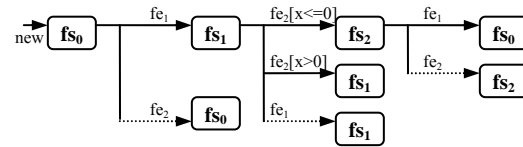


Figure 8. The transition tree for class *Foo*

Testing of integration aspects is somehow different from class testing. An integration aspect involves multiple classes as indicated in its advice models. Advice models can guide aspect implementation. Taking the *FooBar* aspect as an example, its advice model in Fig. 3 or the corresponding part in the woven model in Fig. 5 is typically realized in the aspect code (creating a *Bar* object and then sending a message). Thus test cases for *FooBar* do not need to repeat these operations. However, the states of the *Bar* object are part of the expected result of the test cases and need to be verified against the actual object states achieved by the aspect code. To address this issue as well as maximize the reuse of the above test generation algorithm, our weaving algorithm for test generation transforms advice models by hiding invocations to the constructors and methods of integrated classes and combing the states of the integrated class with base class states to form composite states. A composite state is a state (called primary state) with attachments of the form $\langle precedes/follows S_q \text{ when } e[\phi] \rangle$. Such an attachment means that the primary state of the base class object precedes/follows a sequence of states of integrated classes when it results from transition $e[\phi]$. For example, we reduce the advice model in Fig. 3 to the one in Fig. 9:

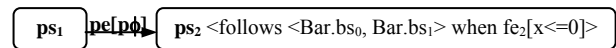


Figure 9. An example of reduced advice model

Such transformation makes the woven models of integration aspects look almost the same as their base class models except for the composite states. Thus, the above test generation algorithm is

applicable to the woven models except for the composite states. To deal with composite states, we modify step (2.1) of the algorithm as follows: if s' is a composite state, then the new node carries the attachment for $e[\phi]$. Steps (2.2) and (2.3) for negative tests are not affected. The composite states are part of the expected result of tests for integration aspects. For *FooBar*, the expected result for the test input ($new, fe_1, fe_2 (-1), fe_1$) is ($fs_0, fs_1, Bar.bs_0, Bar.bs_1, fs_2, fs_0$). Correct implementation of the *FooBar* aspect should produce the state sequence in the specified order.

Normally, an integration aspect does not introduce new states or events to the base and integrated classes. If an integration aspect does not have an *around* type of advice, the woven model is different from the base model only in the composite states. In this case, the integration aspect can reuse the test inputs of all concrete tests for its base classes. However, an integration aspect with *around* advice can update (remove and add) transitions in base class models. The woven model with composite states for the example in Fig. 4(b) is shown in Fig. 10, where “*follows* $\langle Bar.bs_0, Bar.bs_1 \rangle$ when $fe_2[x \leq 0]$ ” is attached to fs_0 . It has subtle differences from *Foo*'s state model in Fig. 1: (1) under state $fs_1, fe_2[x \leq 0]$ no longer results in fs_2 . In fact, no event can reach fs_2 . (2) under state $fs_1, fe_2[x \leq 0]$ results in fs_0 which follows $\langle Bar.bs_0, Bar.bs_1 \rangle$. Nevertheless, any base class test (either negative or positive) starting with $\langle new, fs_0, fe_1, fs_1, fe_2[x \leq 0], fs_2 \rangle$ becomes a positive aspect test $\langle new, fs_0, fe_1, fs_1, fe_2[x=0], Bar.bs_0, Bar.bs_1, fs_0 \rangle$.

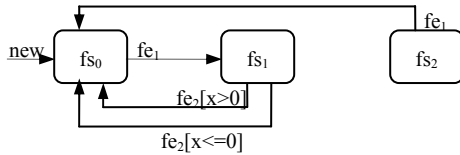


Figure 10. The woven model for *FooBar2*

Reuse of base class tests can help to localize aspect faults. If a base class test passes the base class but fails the aspect-oriented version, then the fault has to do with the integration aspect (provided the integrated classes are also tested adequately).

4. A CASE STUDY

To illustrate our approach, this section introduces testing of integration aspect, *Timing*, in the *Telecom* simulation application and discusses how several types of aspect faults can be revealed.

4.1 Class and Aspect Modeling

The *Connection* class simulates the behavior of physical connection circle units in *Telecom*. It has three states, *pending* (initial state), *completed*, and *dropped*, as shown in Fig. 11. Event *complete* transforms the connection state from *pending* to *completed* and event *drop* from *completed* to *dropped*. Here we revise the original *Connection* code so that it conforms to the model in Fig.11. The testing problem raised by the original code will be discussed in next section. The *Timer* class has two states: *stopped* (initial state or reached from event *stop*) and *started* (reached by event *start*), as shown in Fig. 12.

The *Timing* aspect integrates *Connection* with *Timer* to record connection time for billing purposes. It starts a timer right *after* a connection is completed, and stops the timer right *after* the connection is dropped. Thus two transition join points can be defined: (*pending, complete, , completed*) and (*completed, drop, ,*

dropped). The advice is sending a *start* message and a *stop* message, respectively. The state model for the *Timing* aspect is shown in Fig. 13.

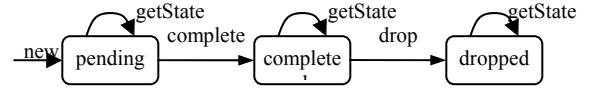


Figure 11. The state model for *Connection*

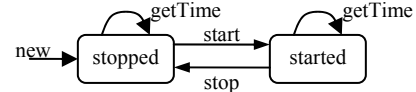


Figure 12. The state model for *Timer*

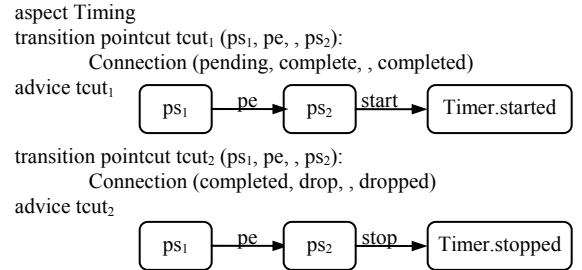


Figure 13. The aspect model for *Timing*

The woven model of the *Timing* aspect is similar to *Connection*'s model except that *completed* and *dropped* are composite states with attachments “ $\langle precedes \langle Timer.started \rangle$ when *complete*” and “ $\langle precedes \langle Timer.stopped \rangle$ when *drop*”, respectively.

To facilitate our discussion, a correct AspectJ implementation for the *Timing* aspect [1] is listed below.

```
public aspect Timing {
    private Timer Connection.timer = new Timer();
    public Timer getTimer(Connection conn) {
        return conn.timer;
    }
    after (Connection c): target(c) &&
        call(void Connection.complete()) {
        getTimer(c).start();
    }
    pointcut endTiming(Connection c):target(c) &&
        call(void Connection.drop());
    after(Connection c): endTiming(c) {
        getTimer(c).stop();
    }
}
```

Listing 1. *Timing* aspect

4.2 Test Generation and Execution

Test generation for the *Timing* aspect followed the aforementioned approach. As all the constructors and methods have no parameters, concrete tests were the same as abstract tests.

One issue for the automated test execution was how to collect the actual states of objects of interest. To address this issue, we designed and implemented a state monitor framework, *StateTracer*, for tracking object states at runtime. *StateTracer* contains an abstract aspect *TraceStateDiagram*, which can be extended to define a concrete trace aspect for a given test task. *TraceStateDiagram* has three abstract elements, *pointcut*

basePointcut(), *String getBaseClasName()*, and *String getStateMethodName()*. The pointcut *basePointcut()* picks out desired join points for comparing state change before and after those join points. The core trace functions are implemented by two piece of advice *before (Object o): actionTrace(o)* and *after (Object o): actionTrace(o)*. *getBaseClasName* returns the base class name, and *getStateMethodName* returns the method name to test current object state. We have tested *StateTracer* and found no side-affect on test execution.

4.3 Detection of Integration Faults

We created several versions of faulty AspectJ code for the *Timing* aspect. Each version indicated one or two specific aspect fault. The faults included incorrect pointcut strength, incorrect advice type, and incorrect advice implementation. Although they are similar to those in incremental modification aspects [13], they indicate incorrect integration.

(1) Pointcut expressions picking out extra join points

One of the common aspect-specific faults has to do with incorrect point expressions that pick out extra join points. A major cause is the inappropriate use of wildcards in pointcut expressions [13]. For the *Telecom* example, we replaced the correct pointcut `call(void Connection.complete())` in the *Timing* aspect with the following:

```
call(void Connection.*());
```

This faulty pointcut picked out all method calls in *Connection* class objects. Thus the timer was started whenever a call was made to any method (e.g. *drop*) of a *Connection* object. Although the correct advice remained unchanged, the aspect did not behave correctly. Consider the following aspect test case: `<new, pending, complete, completed, Timer.started, drop, dropped, Timer.stopped, getState, dropped>`, where *new*, *complete*, *drop*, *getState* are the test input (a sequence of events), and *pending*, *completed*, *Timer.started*, *dropped*, *Timer.stopped*, *dropped* are the expected states for *Connection* and *Timer* objects. Given the test input, the faulty *Timing* aspect actually produced the following object states: *pending*, *completed*, *Timer.started*, *dropped*, *Timer.started*, *Timer.stopped*, *dropped*. There was an extra state *Timer.started* right before *Timer.stopped*. Thus, the fault was revealed. If this fault is not removed from the implementation, the connection time would always be 0.

(2) Pointcut expressions missing join points

A pointcut expression may miss expected join points. For example, we replaced the correct pointcut with the following:

```
call(void Connection.complete())
```

The faulty pointcut did not pick out the expected join point `call(void Connection.complet())` due to the misspelling. For the aspect test `<new, pending, complete, completed, Timer.started, drop, dropped, Timer.stopped, getState, dropped>`, the faulty aspect did not issue the message *Timer.start*. This was observed because it did not produce the *Timer.started* state. Such a fault would lead to incorrect connection time.

(3) Incorrect advice types

Even if pointcuts are implemented correctly, an incorrect advice type may not lead to the expected results. For example, an *after* (or *before*) type may be mistaken as a *before* (or *after*) type. In the *Telecom* application, it requires that the timer be started right *after* the connection is completed, that is, *after* is the correct advice

type. However, an implementation may mistake *before* for *after*, starting the timer *before* the connection is completed. For example one of the faulty *Timing* aspects replaced the two pieces of correct advice with the following (i.e. there were two faults):

```
before (Connection c): target(c) &&
    call(void Connection.complete())
before (Connection c): endTiming(c)
```

For the aspect test `<new, pending, complete, completed, Timer.started, drop, dropped, Timer.stopped, getState, dropped>`, the fault aspect resulted in a wrong order of object states: *pending*, *Timer.started*, *completed*, *Timer.stopped*, *dropped*. Both *Timer.started* and *Timer.stopped* appeared in the wrong position.

(4) Incorrect advice body

Advice implementation may fail to realize the design in the way much like a traditional program does. In our experiment, we created a faulty *Timing* aspect by removing `getTimer(c).start()` and `getTimer(c).stop()` from the two pieces of advice, respectively. Although the pointcuts were correct, the faulty aspect did not start or stop the timer, and thus did not produce the expected object states: *Timer.started* and *Timer.stopped*.

5. DISCUSSION ON DEEPLY CROSSCUT INTEGRATION ASPECTS

Section 4 has described an incremental testing paradigm that first tests base classes (e.g. *Connection*) and integrated classes (e.g. *Timer*) and then tests integration aspects through base classes. If the classes of an aspect-oriented program pass all of their tests but the aspect-oriented program as a whole fail some of the tests, the faults likely have to do with aspects. Before test *Timing* aspect, for example, we test *Connection* and *Timer* with their state models. In this case, however, classes must be tested adequately.

Consider the *Connection* model in Figure 11. The initial state is *Pending*. A *Connection* object is established (i.e. *Completed*) by a method call to *complete*, and then *Dropped* by a method call to *drop*. This is the normal control flow (intended behavior) that a *Connection* object is used. From testing perspective, we need to test not only intended behavior with positive tests, but also unintended behavior with negative tests. In particular, what happens if a *Connection* object receives a *complete* message when it is at the state *Dropped*? This is not defined in the state model – state models are often incomplete. A common understanding is that, for an unspecified event, a program should not change the object state; it may throw an exception or report an error message, though. When the model in Figure 11 is used to test the original *Connection* code in [1], several failures are reported. For example, a method call to *drop* after a *Connection* object is created changes the *Connection* object to *Dropped*; this actual object state is obviously not the expected state *Pending*. A method call to *complete* at the *Dropped* state changes the state to *Completed*.

Figure 14 shows the state model built from the original source code of *Connection* class (the main part is shown in Listing 2, where for discussion purposes, we change *Connection* from an abstract class to a concrete one). The original *Connection* class conforms to this model because it has passed all positive and negative tests generated from this model. However, there are two problems:

- Some transitions, e.g. (*Pending*, *drop*, *Dropped*), are not really meaningful;
- How should the *Timing* aspect advise these questionable operations? Whose fault is it, the *Timing* aspect or the *Connection* class, if timing problems arise?

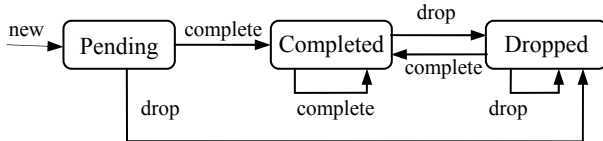


Figure 14. The actual state model for *Connection*

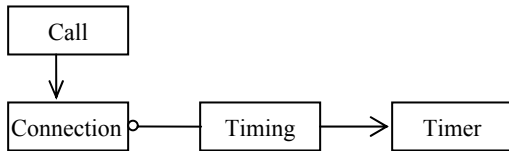


Figure 15. A class hierarchy

Is there anything wrong with the original *Connection* code? Our observation is that nothing is wrong. As a matter of fact, the *Connection* class works in the Telecom simulation. It correctly implements the intended behavior although it does not handle unintended behavior. It is the caller's responsibility to use *Connection* properly, i.e. invoke *new*, *complete*, and *drop* sequentially. On one hand, adequate testing of *Connection* alone does not build our confidence in the program; on the other hand, some negative tests seem unnecessary. To test the *Timing* aspect, we have to bundle *Connection* with its caller, i.e. class *Call*, as shown in Figure 15. This means testing of the *Timing* aspect should be performed through *Call*, including creating a *Call* object and sending messages to the *Call* object, which will trigger the *Timing* aspect. There are two research issues:

- How to model the impacts of *Timing* on *Call* and *Connection* rather than *Connection* alone.
- How to uncover potential aspect faults in *Timing* through testing of *Call*. From the perspective of *Call*, *Timing* is a **deeply crosscut integration aspect**. *Connection* as the base class of *Timing* is a server class of client *Call*.

Therefore, it is desirable to test an integration aspect through the client class of its base class in a class hierarchy when the base class only serves other classes, not user interface. To do so, all the classes involved (client class, server/base class, and integrated class) need to be specified and composed.

```

public class Connection {

    public static final int PENDING = 0;
    public static final int COMPLETE = 1;
    public static final int DROPPED = 2;

    // ...
    private int state = PENDING;

    void complete() {
        state = COMPLETE;
        System.out.println("connection completed");
    }
}

```

```

void drop() {
    state = DROPPED;
    System.out.println("connection dropped");
}
//...
}

```

Listing 2. Part of original *Connection* class

6. RELATED WORK

The weaving mechanism of AOP automatically integrates aspects with classes into an executable whole. It frees the programmer from interleaving different concerns in a monotonous program hierarchy. There are, however, bug hazards with respect to AOP constructs, such as pointcuts and advice.

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 [4]. While this fault model has not yet constituted a fully developed testing strategy, it can provide a useful fault classification for evaluating fault detection ability of testing methods. McEachen and Alexander have explored some of the long-term maintenance issues that can occur with AspectJ. They report that the ability of AspectJ to weave into existing bytecode that already contains woven aspects can create unexpected and potentially unsolvable problems [10].

Zhao et al. have proposed several approaches to unit and integration testing of aspect-oriented programs, such as data flow based approach, system dependence graphs based approach and control flow graphs based approach [14][15][16]. Test cases can be extracted from those graphs. Xie et al. have proposed a framework for generating tests for AspectJ programs, where a wrapper class was created for each base class under test [12]. Zhou et al. have introduced an algorithm based on control flow analysis for selecting relevant test cases [16]. It evaluated test coverage and selected relevant test cases when existing tests could not satisfactorily cover the aspects under test. In [13], we take aspects as incremental modifications to their base classes; we developed an incremental approach to testing whether or not incremental modification aspects and their base classes conform to respective behavior models. This paper deals with a different type of aspect-oriented programs where aspects integrate multiple classes of separate concerns.

Modeling is of critical importance to state-based testing. The main motivation behind modeling is to raise the level of abstraction. A common approach to aspect-oriented modeling (AOM) is to enhance object-oriented modeling with the ability to specify crosscutting concerns. It involves identifying, analyzing, managing, and representing crosscutting concerns. Elrad et al. combines the expressive power of Harel's statecharts with aspect orientation to handle crosscutting concerns [2][3][7]. Coelho and Murphy have developed a tool for presenting crosscutting structures in AspectJ programs [6]. In this paper, we explore state models for specifying integration aspects together with associated classes (base classes, integrated classes, etc.). Our approach is different from other aspect-oriented extensions to state models [2][7]. The latter specifies base state models and aspect state models as different regions of a statechart, where aspects first intercept events sent to base state models and then broadcast the

events to base state models. It relies on a specific naming convention as the weaving mechanism is implicit.

7. CONCLUSIONS

We have presented the state model based approach for testing integration aspects that group separate classes in a larger aggregate. A state-based model of integration aspect captures the expected interaction between base classes and integrated classes. Testing of an integration aspect is performed through the interface of its base classes. It involves verification of composite states, i.e. states of multiple objects, at runtime. As we have discussed, however, it is also desirable to test integration aspects through the clients of their base classes. This requires specification and composition of all classes involved, including client class, server/base class, and integrated class. We expect to solve this problem in the near future.

8. ACKNOWLEDGEMENT

This work was supported in part by the ND NASA EPSCoR through NASA grant #NCC5-582.

9. REFERENCES

- [1] AJDT (AspectJ Development Tools). www.eclipse.org/ajdt/
- [2] Aldawud, O., Bader, F., and Elrad, T. Weaving with Statecharts. *The Second International Workshop on Aspect Oriented Modeling*. 2002.
- [3] Aldawud, T. and Bader, A. UML profile for aspect-oriented software development, *The Third International Workshop on Aspect Oriented Modeling*, 2003.
- [4] Alexander, R. T., Bieman, J. M., and Andrews, A.A. 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] Binder, R. V. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 2000.
- [6] Coelho, W. and Murphy, G.C. ActiveAspect: Presenting crosscutting structure. *ICSE First International Workshop on the Modeling and Analysis of Concerns in Software*. 2005.
- [7] Elrad, T., Aldawud, O., and Bader, A. Expressing aspects using UML behavior and structural diagrams. In *Aspect-Oriented Software Development* (edited by Filman, R.E. et al.). Addison-Wesley, 2005.
- [8] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W.G., An overview of AspectJ. In *Proc. of ECOOP'01*, pp. 327-353, 2001.
- [9] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J. M. and Irwin, J., Aspect-oriented programming. In *Proc. of ECOOP'97*, LNCS 1241, pp. 220-242, 1997.
- [10] McEachen, N. and Alexander, R.T. Distributing classes with woven concerns: an exploration of potential fault scenario. In *Proc. of the Fourth International Conference on Aspect-Oriented Software Development (AOSD'05)*. pp. 192-200, 2005.
- [11] Rinard, M., Salcianu, A., and Bugarara, S. A classification system and analysis for aspect-oriented programs. In *Proc. of FSE'04*, Nov. 2004.
- [12] Xie, T., Zhao, J., Marinov, D., and Notkin, D. Automated test generation for AspectJ programs, *AOSD 2005 Workshop on Testing Aspect-Oriented Programs*, Chicago, 2005.
- [13] Xu, D. and Xu, W. State-based incremental testing of aspect-oriented programs. In *Proc. of the 5th International Conference on Aspect-Oriented Software Development (AOSD'06)*, March 2006, Germany. To appear.
- [14] Zhao, J. and Rinard, M., System dependence graph construction for aspect-oriented programs, *MIT-LCS-TR-891*, Laboratory for Computer Science, MIT, 2003.
- [15] Zhao, J. 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.
- [16] Zhou, Y., Richardson, D., and Ziv, H. Towards a practical approach to test aspect-oriented software. In *Proc. the 2004 Workshop on Testing Component-Based Systems (TECOS)*, Sept. 2004.