State-Based Incremental Testing of Aspect-Oriented Programs

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

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

ABSTRACT
Taking aspects as incremental modifications to their base classes, this paper presents an incremental approach to testing whether or not aspect-oriented programs and their base classes conform to their respective behavior models. We exploit a rigorous aspect-oriented extension to state models for capturing the impact of aspects on the state transitions of base class objects as well as an explicit weaving mechanism for composing aspects into their base models. We generate abstract tests for base classes and aspect-oriented programs from their state models. As base class tests are not necessarily valid for aspect-oriented programs, we identify several rules for maximizing reuse of concrete base class tests for aspects according to the state-based impact of aspects on their base classes. To illustrate our approach, we use two examples that indicate distinctive types of aspect-oriented applications and exhibit fundamental features in complex applications: aspects removing state transitions from base classes and aspects adding and modifying state transitions in base classes. Our results show that majority of base class tests can be reused for aspects, but subtle modifications to some of them are necessary. In particular, positive (or negative) base class tests can become negative (or positive) aspect tests. We also discuss how several types of aspect-specific faults can be revealed by the state-based testing.

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

General Terms
Algorithms, Verification, Design

Keywords
Aspect-oriented programming, model-based testing, incremental testing, state model, aspect-oriented state model.

1. INTRODUCTION
While aspects in aspect-oriented programming (AOP) offer an effective way for modularizing separate concerns, the new programming constructs of AOP languages introduce numerous opportunities for programmers to bring various potential faults with respect to aspects [3]. Generally, an aspect-oriented program consists of aspects and their base classes (or components) that can be woven into an executable whole [14][15]. The base classes in an aspect-oriented program can also be executed independently. From the system architecture perspective, aspects often crosscut multiple base classes. From the base class perspective, however, aspects are essentially incremental modifications to base classes with additional operations and constraints for separate concerns. They provide a paradigm of 'programming by difference', constructing new components by specifying how they differ from the existing components [19]. The incremental modifications of aspects to base classes can impose a significant impact on the object states of base classes. Although aspects in AOP add more code to their base classes, they can not only introduce new object states and transitions, but also remove and update state transitions. As such, aspects may lead to subtle differences in the sequence of messages that can be accepted by the base class objects. In particular, aspect-specific faults likely result in unexpected object states and transitions

To reveal aspect-specific faults, we are motivated to investigate model-based testing, i.e. testing whether or not aspect-oriented programs and their base classes conform to their respective behavior models. Model-based testing is appealing because of several benefits [9][21]: (1) the modeling activity helps clarify requirements and enhance communication between developers and testers; (2) design models, if available, can be reused for testing purposes; (3) model-based testing process can also be (partially) automated; and (4) more importantly, model-based testing can improve error detection capability and reduce testing cost by automatically generating and executing many test cases. Pretschner et al. 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 11% increase in detected errors [22]. Dalah et al. 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 [8]. Using model-based testing methods and tools, Blackburn et al. were able to identify the software error of the Mars Polar Lander (MPL) that is believed to cause the MPL to crash to the Mars surface on December 3, 1999 [5].

In this paper, we present a state-based approach to the incremental testing of aspect-oriented programs, which addresses the following research issues:

- *How to specify the expected impact of aspects on object states for test generation purposes?*
- *To what extent can base class tests be reused for testing aspects? Base class tests are not necessarily valid for testing aspect-oriented programs as aspects likely change transitions of object states.*

AOSD'06, March 20-24, 2006 Bonn, Germany

- *How to determine that a programming fault actually has to do with aspects rather than base classes?*

To capture the expected impact of aspects on the states of base class objects, we exploit aspect-oriented state models, an aspect-oriented extension to state models with testability, for specifying base classes as well as aspects. We compose state models of aspects and base classes by an explicit weaving mechanism and generate abstract test cases from state models for an aspect-oriented program and the corresponding base program. Taking aspects as incremental modifications to their base classes, we identify how to reuse the concrete base class tests for testing aspect-oriented programs according to aspect–oriented state models. Such an incremental approach to testing aspect-oriented programs can significantly reduce testing cost for two reasons: (1) it reuses test cases, the development of which is often an expensive investment; and (2) it helps localize programming problems by identifying aspect-specific faults. For instance, if the base classes of an aspect-oriented program pass all of the state-based tests but the aspect-oriented program as a whole fail some of the tests, the failure would have to do with aspects.

The rest of this paper is organized as follows. Section 2 reviews related work. Section 3 formalizes aspect-oriented state models as well as the weaving mechanism for integrating aspects into base models. Section 4 describes how base class tests can be reused to test aspect-oriented programs. Section 5 discusses how aspect-specific faults may affect object states and state transitions and how they are detected by the state-based incremental testing approach. Section 6 concludes this paper.

2. 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 [3]. This fault model has not yet constituted a fully-developed testing approach. McEachen and Alexander have explored some of the long-term maintenance issues that can occur with AspectJ [16].

Zhao has proposed a data flow based approach to unit testing of aspect-oriented programs [31]. For each aspect or class, the approach performs testing at the intra-module, inter-module, and intra-aspect/intra-class levels. Zhao and Rinard 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 [30]. Control flow graphs were constructed at system and module levels, and then test suites were derived from control flow graphs. The work did not target specific models of most likely faults. To reduce testing cost, Zhou et al. have introduced an algorithm based on control flow analysis for selecting relevant test cases [32]. It evaluated test coverage and selected relevant test cases when existing tests could not satisfactorily cover the aspects under test. Xie et *al*. have proposed a framework for generating test inputs for AspectJ programs, where a wrapper class was created for each base class under test [26]. The above work has primarily focused on generation of unit and integration tests from aspect-oriented programs, whereas our research focuses on whether or not aspect-oriented programs conforms to their behavior models. In [29], we used aspect-oriented UML models to generate tests for exercising aspect-oriented programs. In [28], we presented an approach to state-based test generation for aspect-oriented programs, where aspect-oriented state models were defined in an ad hoc manner and the testing process was not incremental per se. In a recent technical report [27], we formalized an aspect-oriented extension to state models and discussed test generation for aspects-oriented programs. This work was restrictive as it did not associate guard conditions with transitions in state models. In this paper, we have significantly extended our previous work from several perspectives: (1) generalization and formalization of aspect-oriented state models; (2) incremental style of testing that reuses base class tests for aspect-oriented programs; and (3) investigation of how aspect-specific faults would affect object states and state transitions.

2.2 Aspect-Oriented Modeling

With the development of AOP applications, there is an increasing need for addressing crosscutting concerns in the 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, respectively. 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, advices, aspects, and inter-type declarations [2] [10][13][24][25].

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 level of details although their structures may appear to be similar [20]. For example, a traditional state model for design specification does not carry sufficient information for test generation. The testable FREE state model resulted from enhancing a traditional state model with regular expressions [4]. The existing aspect-oriented extensions to state models [1][10] and UML (e.g. [2] [13] [24] [25]) are primarily for the purposes of design specification. Groher and Schulze have investigated AOM for code generation [12]. For program understanding, Coelho and Murphy have developed a tool for presenting crosscutting structures in AspectJ programs [7]. In this paper, we explore aspect-oriented state models for testable specification and test generation of aspect-oriented programs. Our approach is different from other aspect-oriented extensions to state models [1][10]. 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. In comparison, our approach allows to capture the incremental modification nature of aspects and to explicitly specify state and event pointcuts with the support of an explicit weaving mechanism.

3. ASPECT-ORIENTED STATE MODELS

In this section, we first formally define extended state models as a basis for class and aspect specification, and then describe aspect-oriented state models and the weaving mechanism.

3.1 State Models

Objects are encapsulated entities of data and operations that can receive messages from and send messages to other objects [18]. Constraints often exist on the sequence of messages that can be accepted by objects. As these constraints are typically related to object states, state models are a common approach for capturing object behaviors, especially intra-class behaviors. In the following, we extend traditional finite state models as a basis for aspect-oriented state models.

Definition 1 (State Model). A state model M is a 4-tuple (S, E, V, T), where:

(1) S is a finite set of states;

(2) E is a finite set of actions (or events);

(3) V is a finite set of variables;

(4) $T \subseteq S \times E \times \Phi \times S$ is a set of transitions, where Φ is a set of regular logic formula in some language[1]. $(s_i, e, \phi, s_j) \in T$ means that action $e \in E$ transforms state $s_i \in S$ to state $s_j \in S$ under condition $\phi \in \Phi$. ϕ is called the *guard condition* of the transition.

For consistency, if $(s_1, e, \phi_1, s_2) \in T$ and $(s_1, e, \phi_2, s_2) \in T$, then $\phi_1 = \phi_2$. In other words, we do not allow multiple transitions from s_1 to s_2 by the same action e. If action e does transform s_1 to s_2 under different conditions, say, ϕ_1 and ϕ_2, then the transitions can be merged into one with a compound guard condition ϕ_1 *or* ϕ_2. For a state model, we may also specify an initial state $s_0 \in S$. *Definition 1* does not include initial state as part of a state model because state models will also be used to specify aspects. As will be discussed later, the state model for an aspect does not need an initial state. As an aspect-oriented program has a number of state models, we denote the component $X \in \{S, E, V, T\}$ of state model M as $M.X$. It is worth pointing out that a state model can be specified in a table, where each entry (i,j) contains the corresponding action and condition (e and ϕ), if any, that transform state s_i in row i, to state s_j in column j. This makes it convenient to put state models into practice.

In a state model M for class C, events and transitions are related to methods of class C. Specifically, we interpret each transition $(s_i, e, \phi, s_j) \in M.T$ as follows:

- s_i and s_j are abstract states of objects of class C;
- e is corresponding to a method, say $m(\tau_1\ v_1, \tau_2\ v_2, \ldots, \tau_k\ v_k)$, in the specification of class C, where τ_i ($1 \le i \le k$) 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, or explicit parameters v_i ($1 \le i \le k$) of method m. 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.
- (s_i, e, ϕ, s_j) is a call to method m under state s_i that satisfies guard condition ϕ and achieves state s_j.

In addition, a special transition $(, new, \phi_0 , s_0)$ refers to the construction of an object under condition ϕ_0 (ϕ_0 is optional), which results in initial state s_0. Transition $(s_i, e, , s_j)$, where ϕ is omitted, means that the transition is unconditional: any event e (or call to method m) under state s_i results in state s_j. We denote the sequence of transitions $(, new, \phi_0 , s_0), (s_o, e_1, \phi_1, s_1), (s_1, e_2, \phi_2, s_2), \ldots, (s_{n-1}, e_n, \phi_n, s_n)$ by $\langle new[\phi_0], s_0,, e_1[\phi_1], s_1 , e_2[\phi_2], s_2, \ldots, e_n[\phi_n], s_n \rangle$ *or* $\langle new[\phi_0], e_1[\phi_1], e_2[\phi_2], \ldots, e_n[\phi_n] \rangle$. Such a sequence is called an ***abstract test case*** because the parameters of constructor and method calls are not yet assigned specific values.

According to the above interpretation, the specification of object behaviors of a class in our approach actually relies on both the state model and the public interface of the class. The public interface of a class includes complete signatures of the methods. Although instance fields are seldom part of public interfaces, they are often indicated by public *get* methods. For convenience, we allow use of instance fields in state models. This does not lose generality. Note that the state model here is similar to the FREE model [4] except that the guard conditions are explicitly defined with respect to class interfaces.

For example, Fig.1 and Listing 1 show the state model and public interface of the *BankAccount* class, respectively. For clarity, we use b to denote the instance field *balance* and assume that *amt>=0* is a precondition for methods *deposit(amt)* and *withdraw (amt)*. Transition *(Open, withdraw, b-amt>=0, Open)* means that method call *withdraw(amt)* with condition *b-amt>=0* under state *Open* does not change the state.

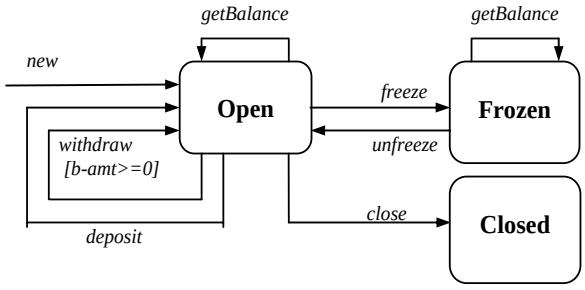


Figure 1. The state model of class *BankAccount*

```
public class BankAccount {
  // constructor, or the new operator
  public BankAccount(double amt);
  // indicating instance field balance - b for short
  public double getBalance();
  public void deposit(double amt);
  public void withdraw(double amt);
  public void freeze();
  public unfreeze();
  public void close();
}
```

Listing 1. The interface of class *BankAccount*

[1] As our focus is testing, this paper uses a programming language, e.g. Java, rather than a formal language.

3.2 Aspect-Oriented State Models

We incorporate aspect-orientation into state models by following the fundamental concepts of AOP, such as aspects, join points, pointcuts, and advices. In our approach, join points can be states, events, or variables in a state model; a pointcut picks out a group of join points; advices are specified as a state model; and an aspect is an encapsulated entity of pointcuts and advice model.

Definition 2 (Pointcut). Pointcuts are defined as follows:

(1) state pointcut <cutname>:<base>.<state>{, <base>.<state>};
(2) event pointcut <cutname>: <base>.<event(paras)> {,<base>. <event(paras)>};
(3) variable pointcut <cutname>: <base>.<variable> {,<base>. <variable>};

where <cutname> identifies a state, event, or variable pointcut; <base> means the state model of a base class; <base>.<state>, <base>.<event(paras)> and <base>. <variable> refers to a state, event, or variable in the base state model, respectively. For convenience, we also reference a pointcut by its name.

Definition 3 (Aspect Model). An aspect model *A* is a 4-tuple *(SP, EP, VP, AM)*, where *SP* is a set of state pointcuts, *EP* is a set of event pointcuts, *VP* is a set of variable pointcuts, *AM = (S, E, V, T)* is a state model (called *advice state model* or simply *advice model*), *AM.S* subsumes all state pointcut names in *SP*, *AM.E* subsumes all event pointcut names in *EP*, and *AM.V* includes all variable pointcut names in *VP* and all variables in the parameters of event pointcuts in *EP*.

We build a model for each aspect. Fig.2 shows the aspect model *Overdraft* that enforces a new banking policy for the base class *BankAccount* in Fig.1. Although it can crosscut other account (e.g. credit card) classes, for simplicity, we specify it only with respect to *BankAccount*. The *Overdraft* aspect allows *one* overdraft as long as the balance is not less than -1000. In the aspect, the states are *Open* (a different name can be used, though) and *Overdrawn*, where *Open* is corresponding to the *Open* state in the base model and *Overdrawn* is a new state. The events are *credit*, *debit* and *get*, which are corresponding to *deposit*, *withdraw*, and *getBalance* in the base model, respectively. The variables used to represent guard conditions are *x* and *b*, which are corresponding to *amt* and *b* in the base model, respectively. Note that the aspect is an addition to the base model as all the transitions from *Open* to *Open* in the base model remain unchanged.

```
aspect Overdraft
state pointcut Open: BankAccount.Open
event pointcut get: BankAccount.getBalance
event pointcut debit(x): BankAccount.withdraw(amt)
event pointcut credit(x): BankAccount.deposit(amt)
variable pointcut b: BankAccount.b
```

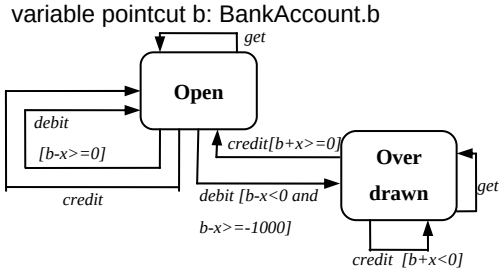


Figure 2. The *Overdraft* aspect

Definition 4. (Aspect-Oriented State Model) An aspect-oriented state model for a system design with *m* classes and *n* aspects is defined by $(\{(BM_i, S0_i)\}, \{A_j\})$ where $S0_i$ is the initial state of base model BM_i and A_j is a state-based aspect model, $1 \le i \le m$ and $1 \le j \le n$.

3.3 The Weaving Mechanism

The semantics of an aspect-oriented state model essentially depends on the weaving mechanism that composes aspect models into base models. As incremental modification to base models, an aspect may affect the base models in various ways, such as:

- adding new transitions among existing states to the base models;
- introducing new states and thus new transitions to the base models;
- removing transitions from the base models;
- modifying guard conditions of transitions in the base models; and
- introducing new events (similar to introductions in AspectJ) to the base models.

For example, the *Overdraft* aspect in Fig. 2 introduces the new state *Overdrawn* and four new transitions. From the base class perspective, the semantics of an aspect model is that the advice model of the aspect overrides the corresponding part in the base model in terms of state, event and variable pointcut specifications. As illustrated in Fig.3, the advice model of *Overdraft* intends to override the transitions associated with the *Open* state in the state model of *BankAccount* with respect to the pointcut specifications. The dashed arrows indicate mappings from states, events and variables in *Overdraft* to those in *BankAccount*. For clarity, not all such mappings are shown. For example, all the three *credit* events in *Overdraft* should be mapped to the same *deposit* event in *BankAccount*. For a transition in the base model, if at most one of the two states in the transition is picked out by some state pointcut in the aspect, then the transition remains unchanged. Examples are *(Open, freeze, , Frozen)*, *(Frozen, unfreeze, , Open)* and *(Open, close, , Closed)*. Fig. 4 shows the woven model of *Overdraft* and *BankAccount*.

Consider another example, *FooBar*, which was discussed publicly at *aspectprogrammer.org* (the AspectJ program can be found in a poster entitled "Aspects as Automaton" at: http://www.aspectprogrammer.org/blogs/adrian/2005/04/aspects_as_auto.html). It is a nice demonstration of AspectJ features. Here we make a change by using states in the base class *FooBar* as well. Fig. 5(a) and Listing 2 are the state model and public interface of *FooBar*. The state model indicates that there is no constraint on the order in which methods *foo* and *bar* are called.

Now we use an aspect to enforce the policy that *foo always comes before bar:* every call to *bar* must be preceded by at least one call to *foo*. After any call to *bar*, *foo* must be called at least once before *bar* can be called again. Fig. 5(b) shows the *Ordering* aspect for the base class *FooBar*. In the advice model, there is no transition from state *S* (i.e. *START* in *FooBar*) to state *B* (i.e. *BAR* in *FooBar*), and no transition from *B* (*BAR* in *FooBar*) to *B*. This implies that the state *BAR* in *FooBar* can only result from a call to the method *bar* from the state *FOO*. As shown in Fig. 5, states *S* and *B* in *Ordering* are mapped to states *START* and *BAR* in *FooBar*, respectively; event *get* in *Ordering* is mapped to event *getS* in *FooBar*; transitions (*START*, *foo*, ,*FOO*), (*FOO*, *bar*,

,*BAR*), (*BAR*, *foo*, ,*FOO*), (*FOO*, *foo*, ,*FOO*), and (*FOO*, *getS*, ,*FOO*) are not affected by the aspect. Fig. 6 shows the *woven model* of *Ordering* and *FooBar*, where the transition from state *START* (or *BAR*) to state *BAR* through event *bar* no long exists. In this example, the *Ordering* aspect only removes state transitions from the base model *FooBar*.

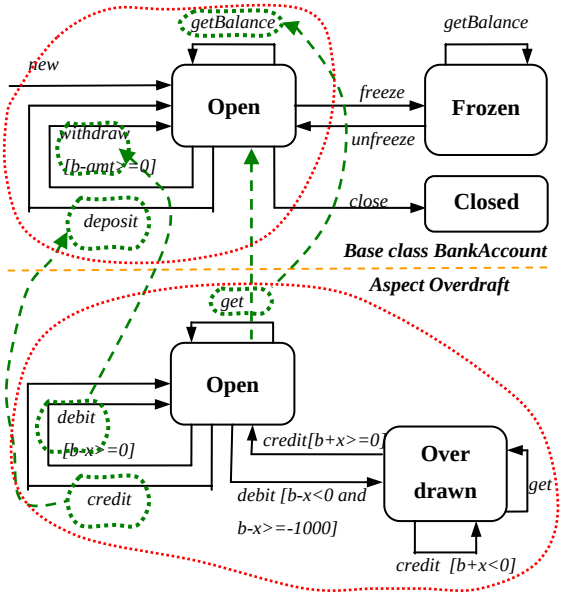


Figure 3. The impact of *Overdraft* on *BankAccount*

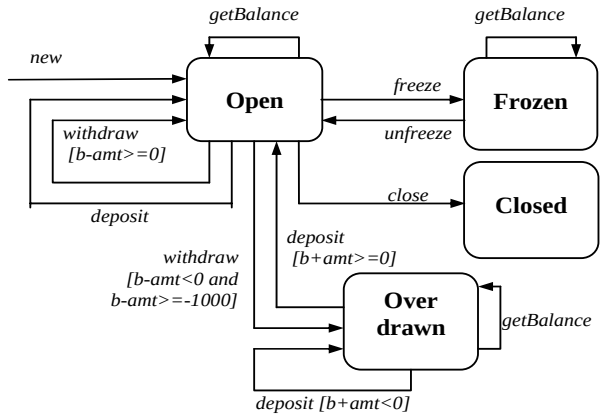
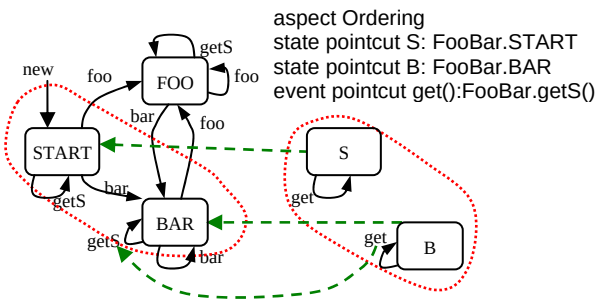


Figure 4. The woven model of *Overdraft* and *BankAccount*



(a) Base model *FooBar* **(b) Aspect *Ordering***

Figure 5. Impact of the *Ordering* aspect on *FooBar*

```
public class FooBar
{    // states: START=0; FOO=1; BAR=2;
  public FooBar();   // constructor
  public void foo();  // set state to FOO
  public void bar();  // set state to BAR
  public int getS(); // return current state
}
```

Listing 2. The interface of class *FooBar*

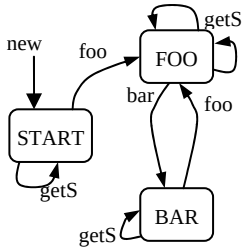


Figure 6. The woven model of *Ordering* and *FooBar*

Now, we formally define the general weaving mechanism for applying an aspect to a base model.

Definition 5 (Weaving Mechanism). Given base model *BM* and aspect model $A = (SP, EP, VP, AM)$, the *woven state model*, *WM*, of weaving aspect *A* into base model *BM* results from the following procedure:

(1) $WM := BM$;
(2) $WM.T := WM.T - \{ (s_i, e, \phi, s_j): (s_i, e, \phi, s_j) \in BM.T, s_i \in A.SP$ and $s_j \in A.SP \}$;
(3) $WM.S := WM.S \cup \{s: s \in AM.S$ and s is a new state$\}$;
(4) $WM.S := WM.S \cup \{e: e \in AM.E$ and e is a new event $\}$;
(5) $WM.T := WM.T \cup \{(s_i, e, \phi, s_j)$: for any $(s_i', ep', \phi', s_j') \in AM.T$, pointcuts s_i', ep', s_j' in *A* picks out s_i, e, s_j in *BM*, respectively, ϕ results from ϕ' by substituting variables in ϕ' for corresponding variables in *BM.V*, where $s_i', s_j' \in A.SP$, $ep' \in A.EP$, $s_i, s_j \in BM.S$, $e \in BM.E\}$;
(6) The initial state of *WM* is the same as that of *BM*.

In *Definition 5*, ":=" refers to the assignment operator. Step (2) says that transition (s_i, e, ϕ, s_j) will not appear in the woven model if both s_i and s_j are included in state pointcuts unless the transition is redefined in the advice model and added by step (5). Steps (3) and (4) add new states and new events in the advice model into the woven net. Step (5) adds all transitions in the advice model to the woven model with corresponding variable substitutions. For other aspects defined on *BM*, we can further apply the weaving process to compose them into the current woven model *WM*. We assume that the order in which aspects are applied is not significant. For an aspect-oriented model $(\{(BM_i, S0_i)\}, \{A_j\})$, we can apply all aspects $\{A_j\}$ to each of the base model BM_i. As such, the whole model of an aspect-oriented system consists of a set of state models $\{(WM_i, S0_i)\}$.

3.4 Discussion

As shown in the *Ordering* aspect of Fig. 5(b), states in an advice model do not need to be strongly connected because an aspect only reflects the incremental modification to its base models. An extreme impact of aspects on base models is that a state in a base model may no longer be reachable – no transition in the woven model can transform any other state into this state. In this case,

we may keep such disconnected states in the woven model. Negative tests with respect to these states are useful for verifying whether or not an aspect-oriented program would reach these states unexpectedly. This will help reveal those aspect-related defects that cause illegal object states.

The weaving mechanism in *Definition 5* indicates that both object-oriented and aspect-oriented systems can be specified by state models. As state-based testing is essentially a black-box technique, it does not care whether the implementation under test is an aspect-oriented or object-oriented program. One could argue that the state model of an aspect-oriented program can be specified without using aspects, i.e., the resulting woven model in *Definition 5* can be defined from scratch. In some sense, the advantages of the AOP's ability to handle crosscutting concerns are not straightforward from the perspective of black-box testing. An aspect-oriented state model, however, can result from aspect-oriented system design that provides guidelines for system implementation. Therefore we can reuse aspect-oriented design models for testing purposes. In addition, aspects in an aspect-oriented state model make explicit the modification to the base state models. This facilitates testing of separate concerns and incremental modification, which are also essential to the aspect-oriented paradigm. As will be discussed later, a test suite for an aspect-oriented program is essentially incremental modification to the test suite for the corresponding base program. This offers a potential of incremental testing – the base classes can be tested before the aspect-oriented version as a whole is available. This also helps localize programming faults by focusing on the base classes first and then on the aspects.

While the aspect-oriented state models in this paper have followed the fundamental concepts of AOP, such as join points, pointcuts, and advices, they represent these concepts in a different way because of the different level of abstraction. In fact, it is not easy to tell from an aspect-oriented state model what the aspect code would look like. State, event, and variable pointcuts do not necessarily have counterparts in AOP programs. Some methods in base classes that are corresponding to event pointcuts are possibly involved in aspect implementation, though. In short, aspect-oriented state models provide a higher level of abstraction than AOP programming, which is desirable for aspect-oriented design.

4. INCREMENTAL TESTING

This section introduces the process of incremental testing, briefly describes test generation from state models, and discusses how to reuse base class tests for aspect-oriented programs as a whole.

4.1 Incremental Testing Process

The general process of our approach to incremental testing of an aspect-oriented program is as follows: (1) build the state models of the base classes; (2) generate abstract test cases from the base models; (3) instantiate the abstract test cases to form concrete test suites for the base classes; (4) test the base classes; (5) build aspect models and weave them into the base models; (6) generate abstract test cases from the woven state models; (7) generate test suites for the aspect-oriented program as a whole by reusing, modifying, and extending concrete base class test cases and instantiate new abstract test cases; and (8) test the aspect-oriented program. Of course, we can combine step (5) into step (1), that is, build complete aspect-oriented models before testing base classes.

4.2 Test Generation from State Models

In our approach, the method for test generation from state models is similar to the modal class test design pattern for object-oriented programs [4], which derives a test suite by transforming a state model to a transition tree and identifying sneak paths with illegal state transitions. The test cases in such a transition tree[2] are primary for the purposes of testing if a program does what it is supposed to do (i.e. positive tests), whereas the test cases represented by sneak paths are for testing if a program does not do what it is not supposed to do (i.e. negative or dirty tests). As this method is in essence a black box technique, it is applicable to test generation from state models of aspect-oriented programs. We slightly enhance this method by integrating sneak paths into conditional transition trees. Given a state model $M = (S, E, V, T)$, with initial state s_0, we transform state model M into a transition tree with sneak paths as follows:

(1) The root node of the transition tree is s_0, the initial state of the state model. We also associate the *new* event and its guard condition with s_0 and mark the root as non-terminal.

(2) For each non-terminal leaf node (say state s_1) in the transition tree, draw a new edge and new node for each event $e \in M.E$.

- If there exists state $s_2 \in M.S$ such that event e transforms s_1 to s_2 under condition ϕ, i.e. $(s_1, e, \phi, s_2) \in M.T$, then the new node represents state s_2. Label the new edge with e as well as ϕ, if any. If state s_2 already appears in the path, mark the new node as terminal, otherwise non-terminal (the new node will be expanded).
- otherwise e is an illegal event at state s_1, label the new edge with e, and assign state s_1 to the new node, and mark it as terminal and negative (i.e. the state remains unchanged if an illegal event happens. This does not mean that event e transforms state s_1 to state s_1).

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

(4) Expand the transition tree using branch coverage for the guard conditions. For each path from the root to a leaf, $<s_0, e_1[\phi_1], s_1, e_2[\phi_2], s_2, \ldots, e_n[\phi_n], s_n>$, start from the root and repeat "finding next conditional transition, say $(s_i, e_i, \phi_i, s_{i+1})$, and creating a new sneak (negative test) path $<s_0, e_1[\phi_1], s_1, \ldots, s_i, e_i[not\ \phi_i], s_i>$ until there is not conditional transition in the path.

Each path from the root to a leaf in a transition tree is an abstract test case. It can be instantiated to derive a number of concrete test cases by assigning constructor and method parameters, if any, specific values that satisfy the guard conditions along the path and by defining the expected outcome. Different traditional test design techniques, e.g. boundary value analysis, can be used for this purpose. For a compound condition in an abstract test, different combinations of truth values for the sub-conditions may be considered (similar to the multi-condition coverage). A preliminary discussion on automated generation of test input for restrictive programs can be found in [28]. As the focus of this paper is on the relations between the state-based test suites of an aspect-oriented program and its base program, the instantiation of

[2] Except for those derived from the extensions of guard conditions, i.e. step (4) in the subsequent algorithm.

abstract tests primarily relies on manual design of test input for each object construction and method call, except for the situations of test reuse.

Fig. 7 shows the transition tree for the woven model of *Ordering* and *FooBar* in Fig. 6. Each path without (or with) dashed edge and node indicates a positive (or negative) test case. The sequence of events in each path is essentially a sequence of object construction and method calls. For example, the path <*new, START, foo, FOO, foo, FOO*> indicates <*new, foo, foo*>. As the transitions in this example have no guard conditions (*foo* and *bar* have no parameters), the test cases derived from all the paths in the tree are actually concrete test cases. They form a test suite for the base class *FooBar*. It is worth pointing out that the two negative test cases for the woven model are as follows:

<*new, START, bar, START*>
<*new, START, foo, FOO, bar, BAR, bar, BAR*>.

The event sequences are <*new, bar*> and <*new, foo, bar, bar*>. These negative cases are critical for detecting the faults in a program that does not enforce the policy that *foo* must come before *bar*.

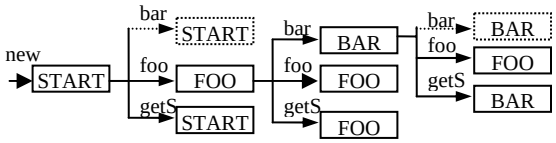


Figure 7. The transition tree for the state model in Fig. 6

To facilitate our discussion on reuse of base class tests, Fig. 8 and Fig. 9 show the transition trees for the base model and woven model of the *BankAccount* example. To save space, '**Part A**' in Fig. 9 is corresponding to that in Fig. 8. Also we use '*' to denote all the events that have not appeared in the sibling nodes. For instance, '*' in the negative test path <*Open, freeze, Frozen, *, Frozen*> in Fig. 8 refers to a number of branches for events *freeze*, *close*, *withdraw*, and *deposit*, respectively. As an example, <*new(500), withdraw(200)*> is a concrete case of the abstract test <*new, Open, withdraw[b-amt>=0], Open*>. A subtlety here is that the conditional event *withdraw[b-amt<0]* in the negative path <*Open, withdraw[b-amt<0], Open*> in Fig. 8 is split into *withdraw[b-amt<-1000]* and *withdraw[b-amt<0 and b-amt>=-1000]* in Fig. 9.

Suppose the number of abstract states and the number of events in a state model are *m* and *n*, respectively, the complexity of the transition tree is $O(m \times n)$. We can further reduce it by removing the accessor methods (events) from state models before generating transition trees. As the implementation of an accessor method is often straightforward, it is easy to test it separately. For instance, *getS* in the above example simply returns the current state. If we are confident in the implementation of *getS*, we can reduce the transition tree in Fig. 7 by removing all the paths that involve *gets*.

4.3 Reuse of Base Class Tests for Aspects

Let us first compare the transition trees of the base model and woven model for the *FooBar* example. They have almost the same paths except for the two negative tests mentioned earlier. These negative tests are actually corresponding to the following positive tests for the base model in Fig. 5(a):

<*new, START, bar, BAR*> and
<*new, START, foo, FOO, bar, BAR, bar, BAR*>

Their event sequences, <*new, bar*> and <*new, foo, bar, bar*>, are the same as those of the negative tests for the woven model. The positive tests for the base model becomes negative ones for the woven model because the aspect has disabled the state-transitions (*START, bar, BAR*), and (*BAR, bar, BAR*). This reflects a close relationship between the test suites for aspect-oriented programs and for their base programs. In other words, testing an aspect-oriented program may make substantial reuse of the state-based test cases of its base classes.

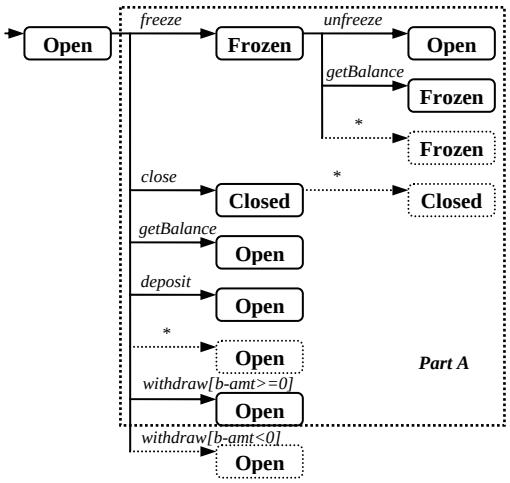


Figure 8. The base model transition tree of *BankAccount*

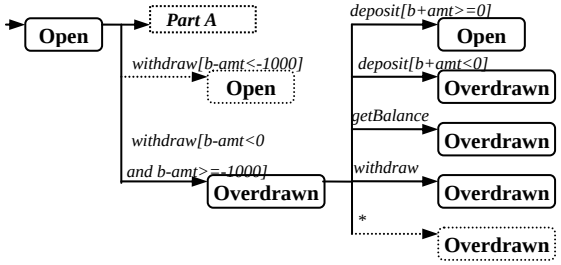


Figure 9. The woven model transition tree of *BankAccount*

Our approach is motivated to maximize reuse of concrete base class tests for aspects. Given base state model *BM=(S, E, V, T)* and aspect *A=(SP, EP, VP, AM)*, *WM* is the woven model of *BM* and *A*. Let Tree(*BM*) and Tree(*WM*) be the abstract test suites for the base model and the woven model, respectively. In the following, we present several rules for reusing base class tests.

Rule 1 *(Full reuse of positive tests)*: For a given positive abstract test <$new[\phi_0]$, s_0, $e_1[\phi_1]$, $s_1 \ldots s_{n-1}$, $e_n[\phi_n]$, s_n> in Tree(*BM*), if (s_{i-1}, e_i, ϕ_i, s_i) $\in$ *BM.T*, (s_{i-1}, e_i, ϕ_i', s_i) $\in$ *WM.T* and $\phi_i \rightarrow \phi_i'$ for any *i* ($0<i\leq n$), then any concrete test of <$new[\phi_0]$, s_0,, $e_1[\phi_1]$, $s_1 \ldots s_{n-1}$, $e_n[\phi_n]$, s_n> for the base class remains as a positive test for the aspect-oriented program.

In *Rule 1*, if an aspect does not modify the guard condition ϕ_i of any event e_i in the path, then $\phi_i' = \phi_i$ and $\phi_i \rightarrow \phi_i'$ holds. In this case, the aspect-oriented program simply inherits the abstract and concrete tests from the base classes. Even if an aspect modifies the guard condition ϕ_i of some event e_i to ϕ_i' such that $\phi_i \rightarrow \phi_i'$, it would not affect reuse of the tests. For a concrete positive test <$new[\phi_0]$, s_0, $e_1[\phi_1]$, $s_1 \ldots s_{n-1}$, $e_n[\phi_n]$, s_n> of base classes where

method parameters are bound to specific values, even if $\phi_i \rightarrow \phi_i'$ does not hold in general, it can still be reused as long as ϕ_i' is true with respect to the given variable bindings. This is similar for other rules. In the following discussion, we focus on reuse of abstract tests, which also applies to reuse of concrete tests. As an aspect may introduce new events to base models, an abstract base class test may be just part of a complete path in the transition tree of the woven model. In this case, the base class test can still be fully used for expansion. For the *FooBar* example, *Rule 1* applies to 5 out of 7 positive abstract tests. For the *BankAccount* example, *Rule 1* applies to all the 6 positive abstract tests.

Rule 2 *(Full reuse of negative tests):* For a given negative abstract test $<new[\phi_0], s_0, e_1[\phi_1], s_1 \ldots s_{n-1}, e_n[not\ \phi_n], s_n>$ in Tree(*BM*), if:

(1) for any i $(0<i\leq n-1)$, $(s_{i-1}, e_i, \phi_i, s_i) \in BM.T$, $(s_{i-1}, e_i, \phi_i', s_i) \in WM.T$, and $not\ \phi_i \rightarrow not\ \phi_i'$ (simply $\phi_i' \rightarrow \phi_i$), and

(2) there does not exist $s \in WM.S$ such that $(s_{n-1}, e_n, \phi, s) \in WM.T$, or there exists $(s_{n-1}, e_n, \phi, s) \in WM.T$ such that $not\ \phi_n \rightarrow not\ \phi$.

then any concrete negative test of $<new[\phi_0], s_0, e_1[\phi_1], s_1 \ldots s_{n-1}, e_n[not\ \phi_n], s_n>$ remains as a negative test for the aspect-oriented program.

A special situation in *Rule 2* is that $<s_{n-1}, e_n, , s_n> \in BM.T$, *i.e. [not* ϕ_n*]* does not occur. Then $not\ \phi_n \rightarrow not\ \phi$ is equivalent to $not\ \phi$. For the *BankAccount* example, *Rule 2* applies to all of the negative base class tests except for those concrete tests of *<new, Open, withdraw[b-amt<0], Open>* such that *b-amt>=-1000*. If *b-amt<-1000*, then the concrete test *<new, Open, withdraw[b-amt<0], Open>* remain as a negative aspect test of *<new, Open, withdraw[b-amt<-1000], Open>*.

Rule 3 *(Modified reuse of positive base class tests as negative aspect tests).* For a given positive abstract test $<new[\phi_0], s_0, e_1[\phi_1], s_1 \ldots s_{n-1}, e_n[\phi_n], s_n>$ in Tree(*BM*), if there exists i $(0<i\leq n)$ such that:

(1) for any j $(0<j\leq i-1)$, $(s_{i-1}, e_i, \phi_i, s_i) \in BM.T$ and $(s_{i-1}, e_i, \phi_i', s_i) \in WM.T$ and $\phi_i \rightarrow \phi_i'$, and

(2) $(s_{i-1}, e_i, \phi_i, s_i) \in BM.T$ and $(s_{i-1}, e_i, \phi, s_i) \notin WM.T$, or there exists $(s_{i-1}, e_i, \phi, s_i) \in WM.T$ and $\phi_i \rightarrow not\ \phi$,

then any concrete test of $<new[\phi_0], s_0, e_1[\phi_1], s_1 \ldots s_{n-1}, e_n[\phi_n], s_n>$ will no longer be a positive test for the woven program. Instead, $<new[\phi_0], s_0, e_1[\phi_1], s_1 \ldots s_{i-1}, e_i[not\ \phi_i], s_{i-1}>$ is a negative test for the aspect-oriented program. Note that the resulting state of the negative test is s_{i-1}, not s_i.

In *Rule 3*, to what extent the test can be reused depends on *i/n*: if *i* is closer to *n*, more reuse is possible. If *i* is close to 1 and *n* is far greater than 1(*n*>>1), then the test is hardly reused. In *FooBar*, *Rule 3* applies to the positive base class tests *<new, START, bar, BAR>* and *<new, START, foo, FOO, bar, BAR, bar, BAR>*. They become the negative aspect tests *<new, START, bar, START>* and *<new, START, foo, FOO, bar, BAR, bar, BAR>*, respectively because transitions *<START, bar, ,BAR>*and *<BAR, bar, , BAR>* no longer exists in the woven model.

Rule 4. *(Reuse for expansion).* For a given negative test $<new[\phi_0], s_0, e_1[\phi_1], s_1 \ldots s_{n-1}, e_n[not\ \phi_n], s_n>$ in Tree(*BM*), if:

(1) for any i $(0<i\leq n-1)$, $(s_{i-1}, e_i, \phi_i, s_i) \in BM.T$, $(s_{i-1}, e_i, \phi_i', s_i) \in WM.T$ and $\phi_i \rightarrow \phi_i'$, and

(2) there exists $s \in WM.S$ such that $(s_{n-1}, e_n, \phi, s) \in WM.T$,

then the test is no longer valid in Tree(*WM*). However, $<new[\phi_0], s_0, e_1[\phi_1], s_1 \ldots s_{n-1}>$ can be further expanded to either positive or negative tests for the aspect-oriented program according to the transitions that transform *s* to other states.

As mentioned earlier, for the *BankAccount* example, a concrete negative base class test of *<new, Open, withdraw[b-amt<0], Open>*, where *b-amt<-1000* also holds, remain as a negative aspect test of *<new, Open, withdraw[b-amt<-1000], Open>*. For a concrete test of *<new, Open, withdraw[b-amt<0], Open>* where *b-amt>=-1000*, however, it can be expanded to derive those negative or positive aspect tests that start with *<new, Open, withdraw[b-amt>=0 and b-amt<-1000], Overdrawn>* (refer to the right-hand side of Fig. 9). In *Rule 4*, another special situation is that *s* already occurs earlier in the path, which means $<new[\phi_0], s_0, e_1[\phi_1], s_1 \ldots s_{n-1}, e_n[\phi], s>$ is a complete test path in the transition tree of the woven model. If it is a positive aspect test, then the negative base class test can be reused with a little additional effort to make the new concrete test satisfy ϕ.

Due to limited space, we will not prove the correctness of the above rules. It can be done by reasoning about aspect-oriented state models and the algorithm of transition tree generation. The rules are useful although, due to the low complexity of the transition tree generation, the transition tree of an aspect-oriented model is constructed from the woven model, rather than obtained by modifying the transition tree of the base model. There are several reasons. Firstly, as the test input and expected outcome of a concrete base class test is provided by testers, rather than automatically generated, we must make best reuse of the concrete tests, which are often an expensive investment. Secondly, the fully reused base class tests, e.g. in *Rules 1* and *2*, are necessary for regression testing of aspect-oriented programs because aspects are incremental modifications to base classes. There is no guarantee that their implementation will not behave unexpectedly. Thirdly, reuse of base class tests helps localize potential faults. If the base classes do not pass a test, then we can focus on the base classes. If the base classes pass the tests but the aspect-oriented program fails some fully reused tests, then we can focus on the aspects. Fourthly, the above rules can be automated by checking aspect-oriented state models and their transitions trees so as to significantly reduce the testing workload. In addition, it is also possible to reuse concrete base class tests that are not derived from the transition trees of state models. Such tests must have followed inadvertently some path in the transition trees provided the system model is sound and complete.

In summary, the state-based test suite of an aspect-oriented program is incremental to that of the base program. Majority of the base class tests can be reused for aspect testing. However, subtle modifications to some of them are required. As in the *FooBar* example, positive base class tests may become negative aspect tests. As in the *BankAccount* example, negative tests may become (part of) positive aspect tests.

5. DETECTING ASPECT FAULTS

A great variety of aspect-specific faults may exist in aspect-oriented programs [3]. Examples include pointcut expressions picking out extra join points, pointcut expressions missing certain join points, incorrect advice types, and incorrect advice implementation. In this section, we discuss how these faults would affect object states and how they can be revealed by the state-based testing approach. While we use the *FooBar* example

for illustration, the situations are similar for the *BankAccount* example. For reference purposes, Listing 3 shows an AspectJ implementation of the *Ordering* aspect.

```
public aspect Ordering {
   pointcut barcut(FooBar fb):
            execution(void FooBar.bar(..)) && target(fb);
   void around(FooBar fb): barcut(fb) {
      if (fb.getS() != FooBar.FOO) {
         System.out.println("bar without foo - illegal");
      }
      else
        proceed(fb);
   }
}
```

Listing 3. Sample AspectJ code for the *Ordering* aspect

We also reduce the *get*S method from the transition tree in Fig. 7. So we have the following tests for the aspect-oriented *FooBar* program.

NTC_1: *new START bar START*
NTC_2: *new START foo FOO bar BAR bar BAR*
PTC_3: *new START foo FOO bar BAR foo FOO*
PTC_4: *new START foo FOO foo FOO*

where NTC and PTC stand for negative test cases and positive test cases, respectively.

(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. For the *Ordering* aspect, one might code the following pointcut:

```
pointcut cut(FooBar fb):
   execution(public void FooBar.*(..)) && target(fb);
```

This pointcut will pick out two join points: execution(public void FooBar.foo(..)) and execution(public void FooBar.bar (..)). The first join point is unexpected. An aspect with such a pointcut will not enforce the ordering policy correctly even if the advice is the same as that in Listing 3. Actually, the above pointcut imposes additional impact on the object states: it requires that both *foo* and *bar* be applied under state *FOO*. Exercising the aspect-oriented program with either PTC_3 or PTC_4 would reveal this fault.

(2) Pointcut expressions missing join points

A pointcut expression may miss expected join points. For example, the following pointcut expression does not pick out the expected join point execution(public void FooBar.bar(..)).

```
pointcut cut(FooBar fb):
      execution(public int FooBar.*(..)) && target(fb);
```

This faulty pointcut will not achieve the expected effect on the object states. The fault can be detected by either negative test case NTC_1 or NTC_2.

(3) Incorrect advice types

Even if pointcuts are specified correctly, a wrong advice type will not lead to the expected results. For example, an after (or before) type may be mistaken as a before (or after) type. For around type advices, proceed() is often used. If an around advice with proceed() is mistaken as a before or after type, it would result in a compilation error. However, proceed() is not always used. For example, the advice in Listing 3 can be implemented in such a way that proceed() is not used at all. In this case, a wrong advice type may have an unexpected impact on the object states. It thus can be detected by the state-based approach.

(4) Incorrect advice implementation

Advice implementation may fail to realize the design in the way much like a traditional program does. For example, the following is a possible fault with respect to the *if* statement in Listing 3:

```
if (fb.getS()==FooBar.BAR)
```

Obviously, fb.getS()==FooBar.BAR is different from fb.getS() != FooBar.FOO due to the fact that an object can be in the *START* state. The faulty condition would allow a transition from state *START* to state *BAR* by a call to method *bar*. This is an unexpected transition. This fault can be revealed by the negative test case NTC_1. Of course, there are other possible faults, such as (fb.getS()!=FooBar.BAR) and (fb.getS()==FooBar.FOO), etc. If a fault results in an unexpected object state like these faults do, exercising the aspect-oriented program with the state-based test cases will reveal it.

6. CONCLUSIONS

We have presented the state-based approach to incremental testing of aspect-oriented programs, which takes aspects as incremental modifications to their base classes. The contribution of this paper is twofold: (1) the rigorous aspect-oriented extension to state models, which facilitates specification of the impact of aspects on the states and transitions of base class objects and generation of abstract test cases, and (2) the investigation of reusing base class tests for conformance testing of aspect-oriented programs. Our work shows that majority of base class tests can be reused for aspects, but subtle modifications to some of them are necessary. In particular, positive (or negative) base class tests can become (part of) negative (or positive) aspect tests. The two examples, *FooBar* and *BankAccount*, indicate two distinctive types of aspect-oriented applications: aspects removing state transitions from base classes and aspects adding and modifying state transitions from base classes. They feature the fundamental impacts that aspects in complex aspect-oriented applications may impose on the states of base class objects.

The incremental testing approach is similar to traditional regression testing. The essential difference is that, aspects as a structured way to specify modifications make it feasible to investigate systematic reuse and modification of the existing tests. Our approach can be adapted to the UML class diagrams and startcharts by using class interfaces, flattening startchart diagrams, and following the convention of guard conditions.

Concerning the future work, we plan to address the following open issues: (1) what kind of base class tests are less likely helpful for revealing aspect faults? (2) how to prioritize the test cases to be reused? (3) how to model and test interference of multiple interacting aspects?

7. REFERENCES

[1] Aldawud, O., Bader, F., and Elrad, T. Weaving with Statecharts. *The Second International Workshop on Aspect Oriented Modeling*. 2002.

[2] Aldawud, T. and Bader, A. UML profile for aspect-oriented software development, *The Third International Workshop on Aspect Oriented Modeling*, 2003.

[3] 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.

[4] Binder, R. V. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 2000.

[5] Blackburn, M, Busser, R., Nauman, A., Knickerbocker, R., and Kasuda, R. Mars Polar Lander fault identification using model-based testing. In *Proc. of the Eighth International Conference on Engineering of Complex Computer Systems*, 2002.

[6] Chavez, C. and Lucena, C. A Metamodel for aspect-oriented modeling, *The Workshop on Aspect-Oriented Modeling with UML*, 2002.

[7] Coelho, W. and Murphy, G.C. ActiveAspect: Presenting crosscutting structure. *ICSE First International Workshop on the Modeling and Analysis of Concerns in Software*. 2005.

[8] Dalal, S. R., Jain, A., Karunanithi, N., Leaton, J. M., Lott, C. M., Patton, G. C., and Horowitz, B. M. *Model-based testing in practice. In Proc. of the 21st International Conf. on Software Engineering (ICSE'99)*, 1999.

[9] El-Far, I. K. and Whittaker, J.A. Model-based software testing. In *Encyclopedia on Software Engineering* (edited by Marciniak), Wiley, 2001.

[10] 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.

[11] Gradecki, J. and Lesiecki, N. *Mastering AspectJ: Aspect-Oriented Programming in Java*. Wiley, 2003.

[12] Groher, I. and Schulze, S. Generating aspect code from UML models. *The Third International Workshop on Aspect-Oriented Modeling*. 2003.

[13] Han, Y., Kniesel, G., and Cremers, A. B. A meta model and modeling notation for AspectJ, *The 5th AOSD Modeling with UML Workshop*, 2004.

[14] 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.

[15] 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.

[16] 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.

[17] Mellor, S. J. A framework for aspect-oriented modeling. *The 4th AOSD Modeling With UML Workshop*, 2003.

[18] Meyer, B. *Object-Oriented Software Construction*. Englewood Cliffs, N.J.: Prentice-Hall, 1988.

[19] Orleans, D. Incremental programming with extensible decisions, In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development*, April 2002, The Netherlands.

[20] Prenninger, W. and Pretschner, A. Abstractions for model-based testing. In *Proc. of the 2nd Intl. Workshop on Test and Analysis of Component Based Systems (TACoS'04)*, Electronic Notes in Theoretical Computer Science 116:59-71, 2005.

[21] Pretschner, A., Prenninger, W., Wagner, S., Kühnel, C., Baumgartner, M., Sostawa, B., Zölch, R., and Stauner, T. One evaluation of model-based testing and its automation. *In Proc. of the 27th International Conf. on Software Engineering (ICSE'05)*, 2005.

[22] Pretschner, A., Slotosch, O., Aiglstorfer, E., and Kriebel, S. Model-based testing for real - The inhouse card case study. *J. Software Tools for Technology Transfer* 5(2-3):140-157, 2004.

[23] Ray, I., France, R., Li, N., and Georg, G. An aspect-based approach to modeling access control concerns. *Information and Software Technology*, vol. 46, no.9, pp. 575-587, 2004.

[24] Stein, D., Hanenberg, S., and Unland, R. An UML-based aspect-oriented design notation for AspectJ. In *Proceedings of the First International Conference on Aspect-Oriented Software Development*, pp. 106–112. ACM Press, 2002.

[25] Tkatchenko, M. and Kiczales, G. Uniform support for modeling crosscutting structure. *The 6th International Workshop on Aspect-Oriented Modeling (AOM'05)*. 2005.

[26] 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.

[27] Xu, D. Test generation from aspect-oriented state models. *Technical Report*, NDSU-CS-TR-XU02, North Dakota State University Department of Computer Science, Sept. 2005.

[28] Xu, D., Xu, W., and Nygard, K. A state-based approach to testing aspect-oriented programs. In *Proc. of the 17th International Conference on Software Engineering and Knowledge Engineering (SEKE'05)*, July 14-16, Taiwan.

[29] Xu, W. and Xu, D. A model-based approach to test generation for aspect-oriented programs. *AOSD 2005 Workshop on Testing Aspect-Oriented Programs*, Chicago, March 2005.

[30] Zhao, J. and Rinard, M., System dependence graph construction for aspect-oriented programs, *MIT-LCS-TR-891*, Laboratory for Computer Science, MIT, 2003.

[31] 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.

[32] 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.