A State-Based Testing Approach for Aspect-Oriented Programming*

CHIEN-HUNG LIU AND CHUAN-WEN CHANG
Department of Computer Science and Information Engineering
National Taipei University of Technology
Taipei, 106 Taiwan
E-mail: {cliu; s4419012}@ntut.edu.tw

In recent years, Aspect Oriented Programming (AOP) has become an emerging technology due to its ability to support the separation of concerns in software development. In particular, AOP allows application requirements to be implemented in separated modules while weaving them together without code tangling. However, this feature also raises a concern about the quality and reliability of AOP programs. Most specifically, the AOP programming constructs, such as join point, pointcut, advice, and aspect, can change the dynamic behavior[1] of original base modules and need to be tested thoroughly to ensure the correctness of AOP programs. In this paper, we propose a state-based testing approach for AOP programs. The approach considers the state-based behavior[2] changes introduced by different advices in multiple aspects. A test model is presented to abstract the state-based behavior of AOP program with the consideration of the interactions between the base modules and aspects. Based on the model, test cases can be derived so as to uncover the potential state behavior errors in the AOP programs. In addition, an example is provided to show the effectiveness of the proposed approach.

Keywords: software testing, aspect-oriented programming, aspect-oriented software development, aspect-oriented modeling, state-based testing, AOP testing, AspectJ

1. INTRODUCTION

Aspect-oriented software development (AOSD) and AOP were first introduced at Xerox PARC in 1990s to empower the programming capabilities of conventional object-oriented programming, especially to support the principle of separation of concerns (SoC) in software development [9, 10]. The main idea of SoC is to modularize the crosscutting concerns of a system. Typically, a concern can be customer required property or technical interest, such as security, that can spans the entire system. Ideally, with SoC, each crosscutting concern can be designed and implemented independently in order to enhance the reusability, extendibility, and maintainability.

Both traditional and object-oriented (OO) programming languages can help programmers in the process of SoC to some extents. For example, procedural programming languages, such as Pascal and C, allow developers to separate concerns into procedures

---

Received February 2, 2007; accepted July 13, 2007.
Communicated by K. Robert Lai, Yu-Chee Tseng and Shu-Yuan Chen.
* This work was supported in part by the National Science Council of Taiwan, R.O.C., under grant No. NSC 95-2221-E-027-092.

[1] The dynamic behavior here means the characteristic that base objects can dynamically change at runtime in respond to events, method invocations, or messages.

[2] The terms "state-based behavior" and "dynamic behavior" are used interchangeably in this paper since the paper adapts state machines to model the dynamic behavior of AOP programs.

while object-oriented programming languages, such as C++ and Java, allow developers to separate concerns into classes and methods. However, the aspect-oriented programming languages, such as AspectJ, take a step further that allow developers to separate crosscutting properties, such as synchronization, memory management, and persistency, that scatter across different procedures (or classes) of a system into standalone code modules called *aspects*.

In particular, the AspectJ, a widely-used AOP programming language for Java, introduces several new programming constructs, such as join point, pointcut, advice, and aspect [1, 8]. A join point is a well-defined point in the program execution flow, such as a method call, a constructor invocation, or a variable access. A pointcut is an expression that specifies a set of join points. An advice is a piece of code that is executed when a join point specified in the pointcut is reached. An aspect is a construct that encapsulates the join point, pointcut, and advice. With AspectJ, the concerns not well captured by traditional programming languages, such as non-functional requirements, can be factored out into aspects in order to achieve the principle of SoC.

The AOSD proposes a new notion to separate crosscutting concerns that can lead to a better architecture design and enable better flexibility and manageability in software development. The aspect-oriented programming languages can facilitate the defining, specifying, designing, and constructing aspects and enforce a better coding style. However, they cannot prevent the errors introduced by the undisciplined programmers or by the misunderstanding of requirements. The new programming constructs and their interactions presented in the aspect-oriented programming languages need to be tested. Most important, the AOP paradigm raises a concern regarding the changes of system behavior caused by the weaving of aspects into the original program. Thus, it becomes a testing challenge to make sure whether the behavior of the woven AOP program aligns to its program specifications.

In this paper, we present a testing approach for AOP programs based on the state-based testing technique. In particular, the approach considers the changes of state-based behavior introduced by the advices in different aspects. A crosscutting model is suggested to represent the aspectual behavior caused by different advice weaving sequences. The state-based behavior of AOP program is then captured using an aspect object state diagram (AsOSD) with consideration of the effects caused by the aspect interactions. From the AsOSD, a test tree can be constructed to generate test cases for uncovering the state behavior errors of AOP programs. In addition, an example is provided to show the effectiveness of the proposed approach.

The rest of the paper is organized as follows. In section 2, we discuss the motivation of our approach through an example. In section 3, a model to abstract different weaving sequences and state-based behavior of AOP programs is presented. Section 4 describes how to generate test cases based on the proposed model. Section 5 discusses the limitation of the presented approach. Section 6 briefly describes the related work and section 7 summarizes our conclusions and future work.

2. A MOTIVATION EXAMPLE

Traditionally, the state-based testing for object-oriented programs considers only the

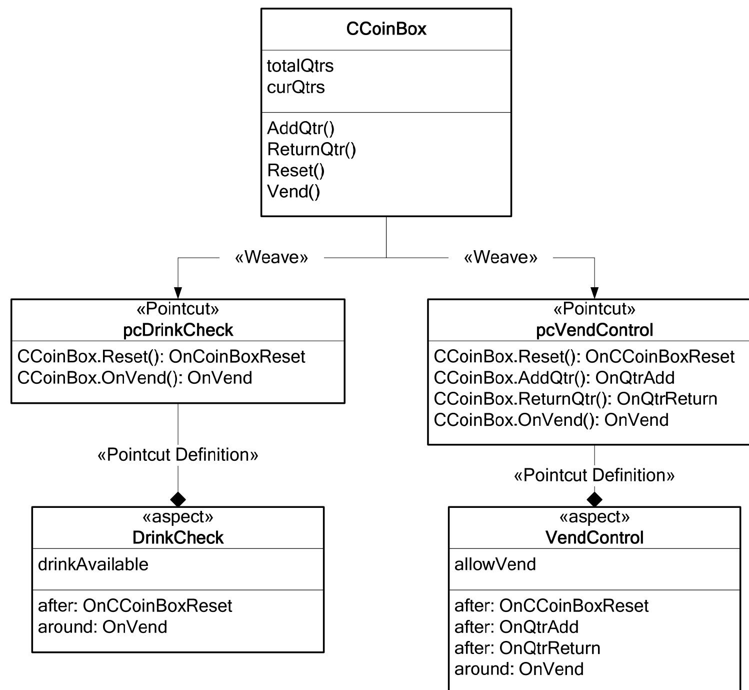


Fig. 1. The class diagram of the auto vending machine example.

interactions between the classes. This may not be adequate for AOP programs since some state behavior errors can happen only in the interactions between the classes and aspects. We will illustrate how such state behavior errors can occur through a simple example.

Consider an auto vending machine example which consists of a coin box and two controlling mechanisms. Fig. 1 shows the class diagram of the example where the coin box is modeled by the base class *CCoinBox* and the two controlling mechanisms are modeled by the aspects *DrinkCheck* and *VendControl*, respectively. The coin box accepts coins inserted by customers and drops a drink when the amount of coins received is enough. The *DrinkCheck* and *VendControl* aspects control the inventory and vending of drinks and are weaved into the *CCoinBox* class through the pointcuts *pcDrinkCheck* and *pcVendControl*, respectively. The separations of *DrinkCheck* and *VendControl* aspects from the *CCoinBox* class allow avoiding invasive composition between the base class and the aspects.

Fig. 2 shows the source code of the vending machine example implemented using AspectJ. To simplify this example, the controls to physical devices of the coin box are omitted. The *CCoinBox* class will keep track of the coins inserted by a customer (denoted *curQtrs*) and the total received coins from all vending transactions (denoted *totalQtrs*). In addition, the *CCoinBox* class consists of the following four major public interfaces in order to perform the basic behavior of an auto vending machine.

(1) *AddQtr*(): Insert a coin into the vending machine. The coins inserted by a customer will be tracked by the internal coin counter *curQtrs*.

(2) *ReturnQtr*(): Return all coins which were inserted by a customer. This method will set the internal coin counter back to zero.
(3) *Reset*(): Bring the vending machine back to the initial state.
(4) *Vend*(): Drop a drink and collect all coins inserted by a customer into its coin storage. Like the real vending machine, the internal coin counter will always be reset back to zero after a valid purchasing since it will return any exceed coins back to customer.

Moreover, the *DrinkCheck* aspect (*i.e.*, *DrinkCheck.aj*) provides the inventory checking control that checks the amount of currently available drinks in the vending machine. It will disable the function of purchasing (the *Vend*() method of the *CCoinBox*

```
01: public class CCoinBox {
02:    private int totalQtrs;
03:    protected int curQtrs;
04:    CCoinBox()   { Reset(); }
05:    void AddQtr()   { curQtrs += 1; }
06:    void ReturnQtr()  { curQtrs = 0; }
07:    void Reset()    { totalQtrs = 0; curQtrs = 0; }
08:    void Vend()     {
09:       totalQtrs = totalQtrs + curQtrs;
10:       curQtrs = 0;
11:       System.out.println("Coke dropped!");
12:    }
13: }
```

(a) *CCoinBox.java*.

```
01: public aspect AspectsInfrastructure {
02:    declare precedence: DrinkCheck, VendControl;
03: }
```

(b) *AspectsInfrastructure.aj*.

```
01: public aspect DrinkCheck {
02:    int drinkAvailable = 0;
03:
04:    pointcut OnCCoinBoxReset():
05:       execution(void CCoinBox.Reset());
06:    pointcut OnVend(CCoinBox cb):
07:       execution(void CCoinBox.Vend()) && this(cb);
08:
09:    //** DC.after(1)
10:    after(): OnCCoinBoxReset(){
11:       drinkAvailable = 3;
12:    }
13:    //** DC.around(2)
14:    void around(CCoinBox cb): OnVend(cb){
15:       if(drinkAvailable > 0)
16:       {
17:          if(cb.curQtrs > 1)
18:             drinkAvailable--;
19:          proceed(cb);
20:       }
21:    }
22: }
```

(c) *DrinkCheck.aj*.

Fig. 2. Source code of the vending machine example.

```
01: public aspect VendControl {
02:    boolean allowVend = false;
03:
04:    pointcut OnCCoinBoxReset() :
05:       execution(void CCoinBox.Reset());
06:    pointcut OnQtrAdd(CCoinBox cb) :
07:       execution(void CCoinBox.AddQtr()) && target(cb);
08:    pointcut OnQtrReturn() :
09:       execution(void CCoinBox.ReturnQtr());
10:    pointcut OnVend() :
11:       execution(void CCoinBox.Vend());
12:
13:    //** VC.after(1)
14:    after(): OnCCoinBoxReset(){
15:       allowVend = false;
16:    }
17:    //** VC.after(2)
18:    after(CCoinBox cb): OnQtrAdd(cb){
19:       if(cb.curQtrs > 1)
20:          allowVend = true;
21:    }
22:    //** VC.after(3)
23:    after(): OnQtrReturn(){
24:       allowVend = false;
25:    }
26:    //** VC.around(4)
27:    void around() : OnVend(){
28:       if(allowVend)
29:          proceed();
30:    }
31: }
```

(d) *VendControl.aj.*

Fig. 2. (Cont'd) Source code of the vending machine example.

class) when all drinks in the storage of the vending machine are sold out. The *VendControl* aspect (*i.e.*, *VendControl.aj*) provides the vending control. It enables the function of *Vend*() method in the *CCoinBox* class when the amount of received coins is enough.

The behavior of the coin box then depends on the interactions between the base class and the two aspects. In particular, the interactions will be affected by the advice ordering. In AspectJ, there are three kinds of advice: *before*, *after*, and *around* which can run before its join points, after its join points, or in the place "around" its join points, respectively. Moreover, more than one advice may apply at a join point. The different advice can come from the same aspect or different aspects. The relative order among the advices is resolved based on advice precedence rules defined in AspectJ [2, 8]. For example, in the same aspect, if both are the *before* advices, then the one that appears earlier in the aspect has precedence over the other that appears later. If the advices are defined in different aspects, the order among the advices will depend on the relationship between the aspects, such as precedence declaration, inheritance, or domination [2, 8].

In the vending machine example, the precedence declaration in *AspectInfrastructure.aj* in Fig. 2 (b) enforces that the advices in the *DrinkCheck* aspect will execute earlier than the advices in the *VendControl* aspect. This precedence declaration is due to the effect that the *Vend*() method should be disabled when all of the drinks stored in the storage were sold out, even the number of coins inserted by the customer was enough.

In Fig. 2, the coin box is implemented in a Java class as a core concern. The vending control mechanisms *DrinkCheck* and *VendControl* are implemented in aspects as crosscutting concerns. Someone may argue that such concern-separation design is unnecessary since the control mechanisms are important for a vending machine and shouldn't be treated as non-functional requirements. However, this design is employed because of two reasons. First, this is just an AOP example used for illustrating the interactions between the base class and aspects. It doesn't matter that the design of functionality is implemented as a core concern or as a crosscutting concern. Second, although the basic idea of AOP is to separate program code into either functional or non-functional categories and then integrates them together, the actual implementation may not restrictedly follow the rule of separation of concerns if it is not clearly defined in the specification.

By carefully analyzing the source code of the vending machine example, one may find a state behavior error existed in the program. This error is not quite obvious since (1) each piece of code (method of the class or advice of the aspects) seems to be implemented following the program requirements; and (2) the program presents correct behavior in normal executing paths (*i.e.*, sunny scenarios). However, the error can cause the vending machine to allow an uncontrolled method call after a specific state. For instance, the state behavior errors can be found in the executing sequence of *AddQtr*(), *AddQtr*(), *Vend*(), and *Vend*(). This sequence represents a scenario that, after inserting two coins and receiving a drink successfully, a customer can get many free drinks without inserting coins anymore till the vending machine is out of stock. This error is introduced by an incorrect state transition and can be removed by changing the state of variable *allowVend* to *false* in the *after*() : *OnVend*() advice after a successful purchase. Fig. 3 shows the advice after removing the error.

```
26:     //** VC.around(4)
27:     void around() : OnVend(){
28:         if(allowVend)
29:         {
30:             proceed();
31:             allowVend = false;
32:         }
33:     }
```

Fig. 3. The *after*() : *OnVend*() advice after removing a bug.

3. THE STATE-BASED TESTING APPROACH FOR AOP

In this section, we present an approach for testing AOP programs based on the program specifications or source code. The approach consists of several steps in order to generate test cases for uncovering state behavior errors. Basically, the first step is to identify the state variables and transitions through analyzing the AOP specification or source code. The second step is to capture the possible transition changes of each state transition caused by aspects. Finally, by integrating the transitions of state variables, a test tree is constructed so as to derive test cases for the AOP program. We will illustrate the details of these steps in the following sections.

3.1 The State Variable and Transition Analysis

In general, the state-based behavior of OO programs can be represented using state-transition diagrams or UML statecharts (or other variants) [5, 12]. In this paper, we adapt the state-based behavior representation presented in [11] for AOP programs. Basically, in [11] the state-based behavior of the OO program is represented using a set of composite and concurrent state machines, where each state machine representing the behavior of a single state variable. In particular, the state variable can be the data members of an object and the transition can be the member functions of the object. The behavior of the OO program is modeled as the interactions among the set of state machines.

To represent the state-based behavior of AOP program, we need to identify the *state variables* and *transitions* of the AOP program. Unlike OO program, the behavior of AOP program can be affected by both the objects and aspects since the woven aspects become an extension of the objects. Thus, the data members and the member functions (*i.e.*, *advice*) of aspects need to be considered in capturing the state-based behavior of the AOP program [2, 8]. Specifically, in our representation, the data members of objects and aspects involving state-based behavior are considered as state variables. The identification of such data members usually requires human interventions. The following steps give some suggestions for identifying the candidate state variables of an object or an aspect.

(1) For each object or each aspect, identify the data members whose values can affect the control logic of methods, method or advice invocation sequences, output events/messages, or observable behavior of the object or the aspect.
(2) For each aspect, consider also the data members defined using *inter-type declaration* mechanism [2, 8].
(3) If the above identified data member is an atomic entity (*i.e*., simple data type, such as integer, boolean, real, or enumerate), the data member can be considered as a candidate state variable.
(4) If the data member is a composite entity (*i.e*., user-defined type or class), decompose this entity recursively until it contains only atomic entities. Verify all the atomic entities to see if they can be candidate state variables based on the ideas suggested in step (1).
(5) The candidate state variables for the object or the aspect can be those identified in steps (3) and (4).

To capture the state transitions, the member functions of objects and aspects are treated differently in the proposed representation. Unlike the object's member function that can be invoked directly and individually by an external client, the aspect's advice (*i.e*., member function) is usually invoked indirectly through the member function of an object. Most important, the advices of an aspect cannot be invoked individually. Instead, they will be triggered automatically in a single invocation of the aspect according to the rules of advice ordering and the values of state variables (section 3.3 will provide more detailed description). Thus, in identifying the state transitions of AOP program, we consider the member functions of objects and the possible member function calling sequences of aspects.

Table 1. The state variables and transitions of the vending machine example.

(a) State variables.

State variables	Value domain
CCoinBox.curQtrs	≤ 0, > 0 and ≤ 1, > 1
DrinkCheck.drinkAvailable	> 0, ≤ 0
VendControl.allowVend	false, true

(b) State transitions obtained from *CCoinBox*.

State transitions
CCoinBox.AddQtr()
CCoinBox.ReturnQtr()
CCoinBox.Reset()
CCoinBox.Vend()

Consider the source code of the vending machine example in Fig. 2. There are four data members: *totalQtrs*, *curQtrs*, *drinkAvailable*, and *allowVend*, in the base class and aspects. By analyzing the source code, only the values of *curQtrs*, *drinkAvailable* and *allowVend* can affect the behavior of the vending machine and can be considered as state variables. The possible values and ranges of these state variables are given in Table 1. The value domain of each state variable can be obtained by using partition analysis to identify the subdomains of the variable in which the AOP program has different behavior or executes different control-flow paths [7]. Moreover, Table 1 lists the possible state transitions that are obtained from the *CCoinBox* class. These transitions can be invoked by external clients. The transitions resulted from *DrinkCheck* and *VendControl* aspects will be discussed in section 3.3.

3.2 The Object State Diagram

The Object State Diagram (OSD) [11] is a set of concurrent and communication state machines that can be used to represent the state-based behavior of an OO program. In OSD, the behavior of an individual state variable is modeled using an atomic OSD. The behavior of an object is represented using a composite OSD that is an aggregation of all atomic OSDs of the state variables of the object. The overall behavior of an OO program is then represented in terms of the interactions among the composite OSDs of the objects in the program. The OSD allows testers to have a deeper understanding about the state transiting of each individual state variable. In this paper, we adapt OSD to represent the state-based behavior of an AOP program before the effects of aspect weaving are considered. The following steps briefly outline the procedure to construct the OSD based on the identified state variables and transitions. A detailed construction algorithm can refer to [11].

(1) For each value domain of a state variable, place an oval in the diagram with a label indicating the value domain. Each oval represents a possible state of the state variable.
(2) Identify the initial state for each state variable. The initial state is indicated by using an arrow with a filled black circle pointing to the state.

(3) For each state S created in step (1), examine the transitions one by one. Let t be the transition under examination.
(4) If transition t can cause a state change from S to another state S', draw an arrow from S to S'; otherwise, draw an arrow form S back to itself. Label the arrow with the name of transition t.
(5) Check if there is any condition c that must be satisfied before transition t can happen. If yes, let c be the guard condition [5] of t and change the label of the transition to $[c]t$. A transition can happen only if its guard condition is satisfied.
(6) If the execution of transition t can cause another transition t' to happen, let t' be the trigger of t and change the label of the transition to $[c]t/t'$.
(7) Repeat from step (3) until all transitions for each state are examined.

Based on the state variables and transitions in Table 1, the OSD for the vending machine example can be constructed as shown in Fig. 4. It should be noted that Fig. 4 does not show the transitions for those states derived from the data members of aspects. This is because those states can be changed only through a series execution of woven advices, not by a single member function of the object or the aspect. We will show how to obtain the transitions for these states in sections 3.3 and 3.4.

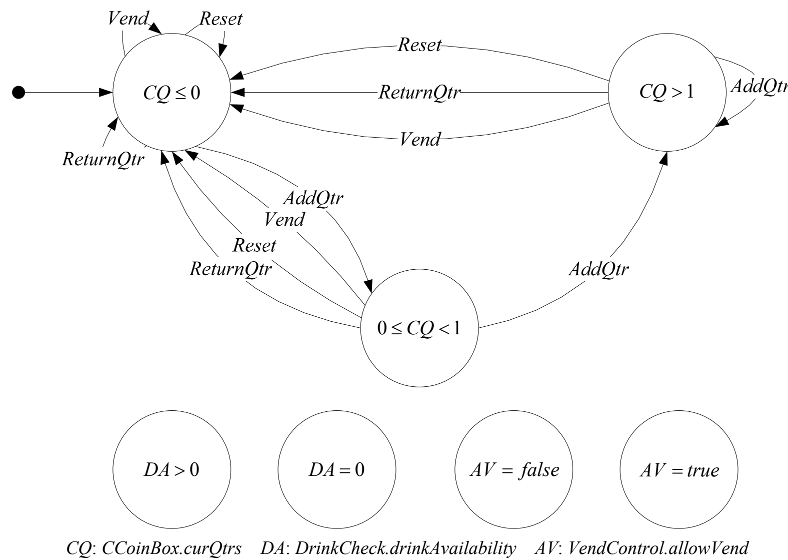


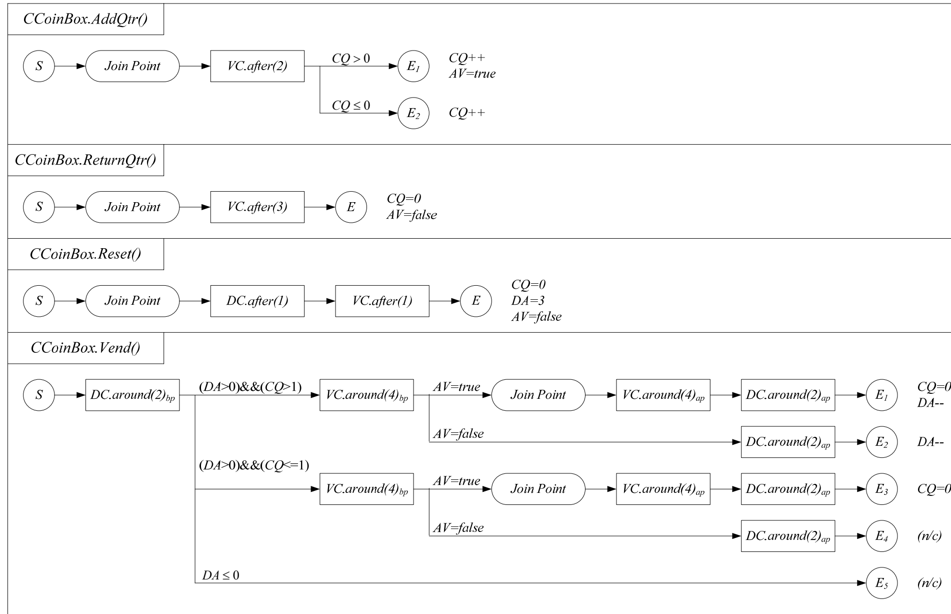
Fig. 4. The object state diagram for the vending machine example.

3.3 The Crosscut Weaving Model

As mentioned in section 3.2, the advices of an aspect will be called internally and automatically when the join point (a member function of the class) is invoked by external clients. In particular, the execution order of the advices depends on how the advices are weaved. For a single-advice woven scenario, the execution order of the advices can be (1)

the *before* advice followed by the join point; (2) the join point followed by the *after* advice; or (3) the *around* advice around the join point. Note that the *around* advice can invoke the join point during its execution depending on whether the function *proceed*() is called or not. For a multi-advice woven scenario, the typical order of the advices will be the set of *before* advices, the set of *around* advices, followed by the set of *after* advices. The detailed ordering rules of advices can refer to [2, 8].

Since the invocation of the join point can cause a series *before*, *after*, or *around* advices to be executed, the original execution result of the join point can be affected by the advices. This means that the state variables can be influenced by various advices and by the execution sequences of the advices. In order to obtain how the values of state variables are changed, a *crosscut weaving model* (*CWM*) is proposed. The model considers the possible execution results of the advices and their overall effects to the state variables. Basically, in this model the execution order of a join point and its weaved advices is first examined based on the advice precedence rules. The execution paths of each advice are then depicted one by one in order to obtain the possible values of state variables after weaving the advices into the join point. Notice that an *around* advice may or may not invoke the join point. In such a case, the execution paths of the *around* advice need to be analyzed and the join point will be added into the paths according to the logic of the advice.



***Advice*:** *DC*: *DrinkControl* *VC*: *VendControl*
***State Variable*:** *CQ*: *CCoinBox.curQtrs* *DA*: *DrinkCheck.drinkAvailability* *AV*: *VendControl.allowVend*

Fig. 5. The CWM for the vending machine example.

Fig. 5 shows the CWM for the vending machine example. In the figure, there are four join points. Each join point is weaved into corresponding advices according to the

advice precedence rules. The execution paths of each advice are depicted and the values of state variables after the paths being executed are also denoted. For example, the join point *CCoinBox.AddQtr*() is added before the advice *VC.after*(2). In addition, by analyzing *VC.after*(2), two execution paths are existed and the values of state variables *curQtrs* and *allowVend* will be changed after the paths being executed (denoted *CQ* and *AV* at the end of the execution paths in the figure).

Moreover, the join point *CCoinBox.Vend*() is weaved into two aspects *DrinkCheck* (denoted as *DC*) and *VendControl* (denoted as *VC*). Since *DC* has a higher precedence than *VC*, the join point *CCoinBox.Vend*() will be added into *VC.around*(4) according to the multi-advice ordering rule [2, 8]. Further, by examining the *DC.around*(2) advice, we can obtain three possible execution paths in which two of them will invoke the advice *VC.around*(4). Similarly, the *VC.around*(4) has two execution paths in which one will invoke the *CCoinBox.Vend*() depending on whether or not the *proceed*() function is called in the path.

The following steps briefly outline the procedure to construct the CWM for a join point:

(1) Create a start point (denoted as a circle-s) in the diagram. The start point represents the start of the execution paths for the join point.
(2) Based on the advice precedence rules, add the first member function (can be a *before* advice, an *around* advice, or a join point) that can be invoked into the path. Denote this new added advice or join point as *C* which will be the next member function to be examined. Add an arrow from the start point to *C*.
(3) If *C* is a joint point, add the next advice *C′* that can be invoked right after *C* based on the advice precedence rules. Add an arrow from *C* to *C′* and let *C* = *C′*.
(4) For each control-flow path of *C*, check if a joint point or an advice (denoted *C′*) can be invoked through the path based on the advice precedence rules.
    (a) If not, add an end point (denoted as a circle-e) to the end of the execution path. Label the resulting values of the state variables after symbolically executing the path from the start point to this end point.
    (b) If yes, add *C′* into the execution path according to the control structure of *C*. Add an arrow from *C* to *C′*.
    (c) If *C′* is a joint point and no advices can be invoked around or after *C′*, add an end point after *C′* and label the values of state variables similar to step 4(a). Otherwise, add the next advice *C″* that can be invoked into the execution path based on the advice precedence rules. Add an arrow from *C′* to *C″* Let *C* = *C″* and expand further.
    (d) If *C′* is not a joint point, let *C* = *C′* and expand further.
(5) Repeat from step (4) until no expansion is possible (*i.e.*, every path is ended with an end point).

It should be pointed out that the above CWM is not intended to be a comprehensive model and is not fully compliant to all the features of AspectJ. However, this model has captured the general weaving concepts of AOP (*e.g.*, before, around, and after). Other advanced features, such as aspect inherence, that are implemented on specific AOP programming languages will be considered in the CWM in our future work.

3.4 Aspect Object State Diagram

The crosscut weaving model shows the effects to the join points after advice weaving. From the model, we can obtain that, after weaving, (1) the join points may not be executed; (2) the original effects of the join points to the object's data members (i.e., state variables) may be changed by the advices; and (3) the values of some aspect's data members can be changed by advices. Therefore, in order to take into account the weaving effects, the OSD for an AOP program before aspect weaving needs to be adapted so as to represent the state-based behavior after aspect weaving. To distinguish the state-based behavior before and after aspect weaving, the OSD for an AOP program after aspect weaving is called *Aspect Object State Diagram* (*AsOSD*).

Basically the AsOSD can be constructed in three steps by analyzing the CWM and the OSD before the aspect weaving. The first step is to identify the transitions for aspect's state variables. From each execution path of the join points in the CWM, the aspect member functions (*i.e.*, transitions) and their results can be easily identified. For instance, consider the execution paths of the *CCoinBox.AddQtr* join point in Fig. 5. There is a transition *VC.after*(2) that will change the state variable *AV* to *true*. Fig. 6 shows the transitions of aspect state variables for the vending machine example obtained from the CWM.

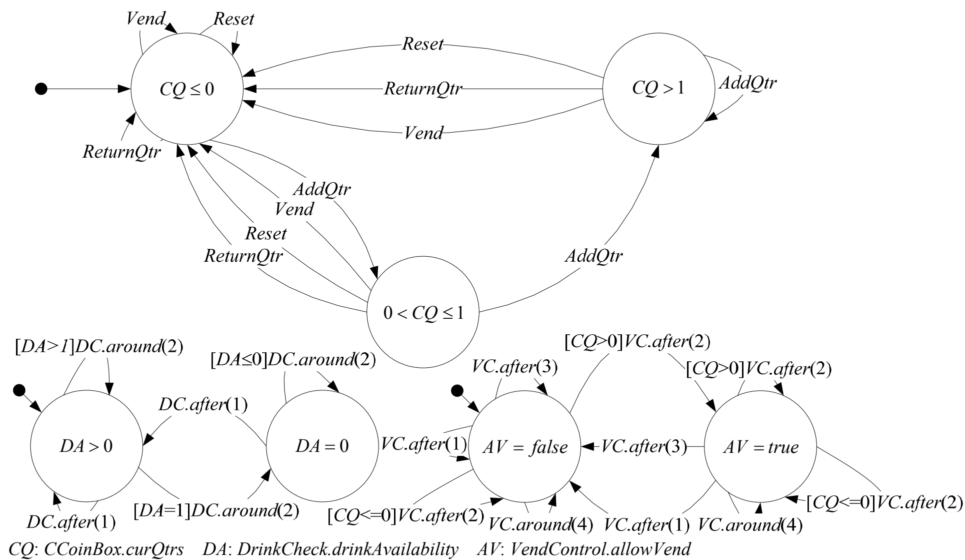


Fig. 6. The AsOSD showing transitions for aspect state variables.

Notice that, in some situations, depending on the values of state variables, a member function may or may not cause a state change. For example, in Fig. 5 the execution of the advice *DC.around*(2) can cause the value of state variable *DA* to be decreased by one (*i.e.*, *DA*--) when the condition *DA* > 0 is satisfied. Thus, the advice *DC.around*(2) can cause a transition from the state *DA* > 0 to the state *DA* = 0 under the condition *DA* = 1; otherwise, the state will remain at *DA* > 0. In such a case, a guard condition can be im-

posed on the transition to resolve this undeterministic problem. As shown in Fig. 6, the advice *DC.around*(2) at state $DA > 1$ can cause two possible transitions and they are distinguished using the guard conditions [$DA > 1$] and [$DA = 1$], respectively.

The second step is to identify the transitions of object state variables that are influenced by aspect weaving and the variants of the transitions. From the CWM in Fig. 5, we can find that the original four transitions *CCoinBox.AddQtr*(), *CCoinBox.ReturnQtr*(), *CCoinBox.Reset*(), and *CCoinBox.Vend*() are expanded to nine execution paths, where each path represents a possible transition variant to the original transition. Here, a transition variant for a transition T is denoted by [*cond*]T, where [*cond*] is the branch condition (or guard condition) of the execution path.

For example in Fig. 5, there are two execution paths for the *CCoinBox.AddQtr*() function. Each path has different guard condition and result. This means that, after weaving, the original transition *CCoinBox.AddQtr*() will turn into two variants that can change the effects to the state variables different from those caused by the original function *CCoinBox.AddQtr*(). Note that in Fig. 5 there are three paths in which the join points are not invoked. This indicates that the weaving will cause the effects of the original transitions to be ignored.

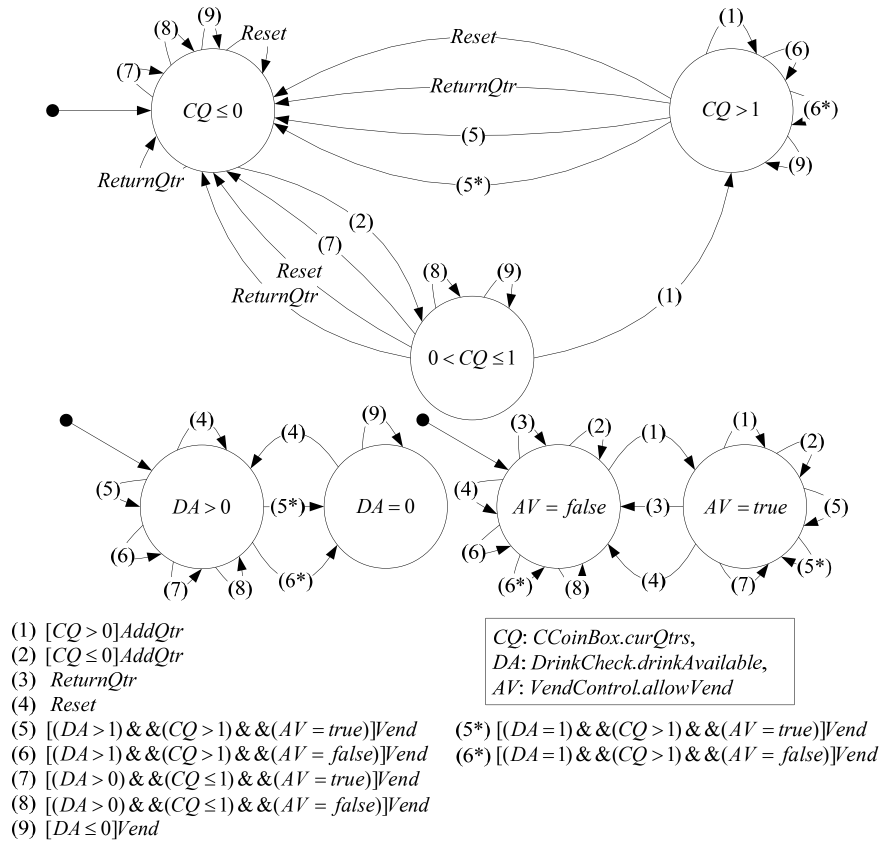


Fig. 7. The AsOSD for the vending machine example.

Moreover, there are two paths in the CWM that can cause the value of the state variable *DA* to be decreased by one (*i.e.*, *DA*--). As described above, the execution result of *DA*-- may or may not cause a state change and two guard conditions are required to resolve this situation. This means that the two execution paths actually can result in four possible transition variants depending on the value of the state variable *DA*. As shown in Fig. 7, the AsOSD for the vending machine example, the transition variants (5) and (6) represent the paths resulted to state $DA > 0$ while the transition variants (5*) and (6*) represent the paths resulted to state $DA = 0$.

Based on the transition variants obtained in previous step, the last step is to replace the original transitions of AsOSD with appropriate transition variants. For the transitions of object state variables, this can be done by replacing the original transitions with proper transition variants via analyzing the values of corresponding state variables and the guard conditions of the variants. For example, the original transition *Vend*() has five transition variants (5) to (9) as shown in Fig. 7. However, by analyzing the state values of *CQ* and the guard conditions of the variants, not all of the variants can be invoked for each state of *CQ*. For instance, in the state $CQ \leq 0$, the transition *Vend*() can be replaced by its variants (7) to (9). The variants (5) and (6) are invalid since the state value $CQ \leq 0$ does not satisfy their guard conditions.

For the transitions of aspect state variables, the original transition caused by an advice can be replaced with proper transition variants that invoke the advice by analyzing the values of corresponding state variables and the guard conditions of the variants. For example, the transition [$CQ > 0$]*VC.after*(2) can be replaced by [$CQ > 0$]*AddQtr*() since *VC.after*(2) is triggered by *AddQtr*() under the condition $CQ > 0$. Similarly, the transition *VC.after*(1) can be replaced by *Reset*() since *VC.after*(1) is invoked by *Reset*(). Fig. 7 shows the AsOSD obtained from Fig. 6 by replacing the original transitions with proper transition variants.

4. TEST CASE GENERATION

Based on the AsOSD, a test tree for an AOP program can be constructed. The test tree is presented in a spanning tree and consists of a number of nodes and links [11]. Each node in the test tree represents a state of the AOP program, and each link represents a state transition from a source state to the transited state through a valid transition (*i.e.*, object member function). In our test tree, each node, denoted $S_n(i, j,.., k)$, is a composition of state variables and represents a state of AOP program. The sub-index n is a unique identifier that indicates a specific state in the test tree. Each vector within the parenthesis is the composition of the value domain indexes for corresponding state variables. The following steps briefly outline the procedure of constructing the test tree for AOP programs.

(1) Based on the AsOSD, create an initial state as the root of the test tree. The initial state, denoted $S_1(IS_1, IS_2, \ldots, IS_n)$, can be identified as the composition of the initial states of all state variables, denoted IS_i, $i = 1, 2, \ldots, n$. For instance, in the vending machine example, the initial state is $S_1(CQ \leq 0, DA > 0, AV = false)$ which is the composition of the state variables *CQ*, *DA*, and *AV*. The values of the state variables are $CQ \leq 0$, $DA > 0$, and $AV = false$, respectively.

(2) In the following steps, each node of the test tree has a composite state denoted $S_n(i, j, \ldots, k)$, where $i, j, \ldots, k$ are the atomic states of corresponding state variables. Examine the nodes in the tree one by one starting from the root.
(3) Let $S_n(i, j, \ldots, k)$ be the node under examination. If there is a transition $[c]t$ in a base class that can lead state j to j', and state $S_n(i, j, \ldots, k)$ satisfies condition c, then attach a branch with a label $[c]t$ and a successor node $S_{n+1}(i, j', \ldots, k)$ to the node $S_n(i, j, \ldots, k)$.
(4) If transition $[c]t$ is also in aspects and $S_n(i, j, \ldots, k)$ satisfies c, update the values of corresponding aspect state variables in the successor node $S_{n+1}(i, j', \ldots, k)$ accordingly.
(5) If transition $[c]t$ also triggers other transitions in a base class, update the values of corresponding object state variables in the successor node $S_{n+1}(i, j', \ldots, k)$ accordingly.
(6) If the node $S_n(i, j, \ldots, k)$ has already occurred at a higher level in the tree, then the node becomes a leaf node of the tree and will not be expanded further.
(7) Repeat from step (3) until no expansion is possible.

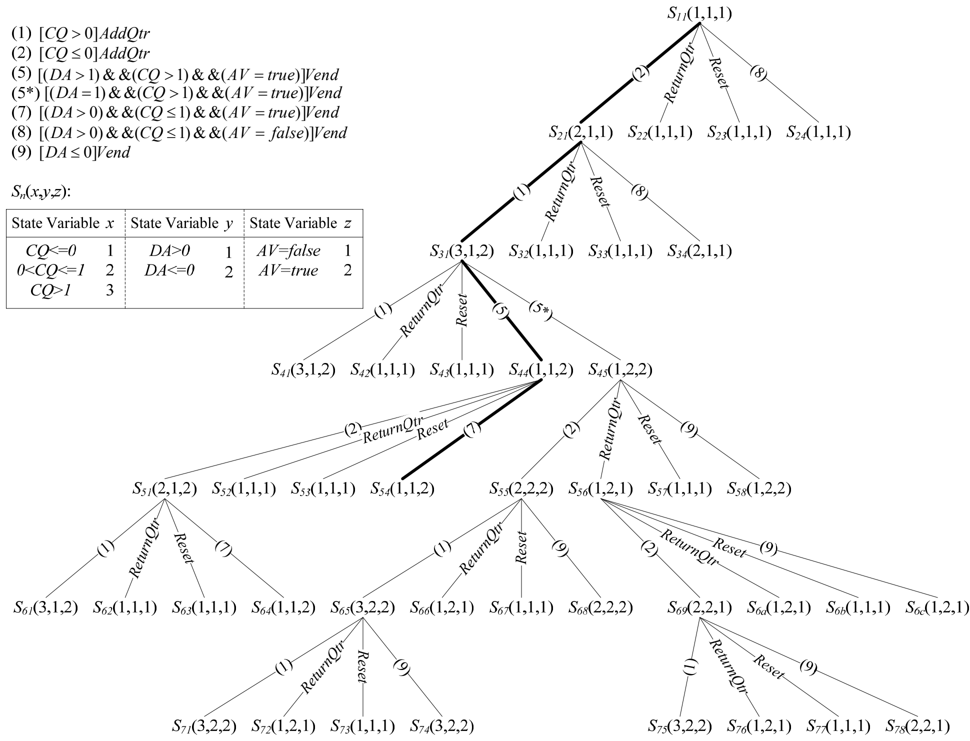


Fig. 8. The test tree for the vending machine example.

Fig. 8 shows the test tree for the vending machine constructed using the above procedure. From the test tree, we can derive test cases for testing AOP programs. Basically, a test case will be a path from the root node to any leaf node of the test tree. The test case represents a sequence of member function calls and can be useful for detecting the state

behavior errors of AOP programs. For example, from the test tree, a state behavior error of the vending machine program can be revealed by the test case derived via the traversing path from node S_{11} to node S_{54}. The test case represents a series function calls *AddQtr*();*AddQtr*();*Vend*();*Vend*() by an external client to the *CCoinBox* object. After executing this test case, the state of the vending machine will change to ($CQ \leq 0$, $DA > 0$, $AV = true$) which is an incorrect state as described in section 2, and hence, the bug of the vending machine can be detected.

It should be noted that the number of test cases derived from the test tree can increase rapidly as the size of the tree becomes large. To avoid excessive test cases while achieving adequate test, we can employ test coverage criteria. There are several state-based test coverage criteria [4, 15] that can be applied to the proposed approach. For example, the transition, full predicate, and transition-pair coverage criteria can be useful to select a subset of test cases generated from the test tree. The transition coverage criterion requires each possible transition to be covered by the test set. The full predicate coverage criterion requires the test set to cover the tests that each clause in the predicates (*i.e.*, guard condition) of the transitions to take on the values of TRUE and FALSE. In addition, the transition-pair coverage criterion requires that each pair of adjacent transitions is covered in the test set.

5. DISCUSSIONS OF THE APPROACH

In this section, we describe the usefulness of the proposed testing approach. Several limitations and possible alleviations of the current approach are also discussed. In general, the AOP programs can be considered as an extension of object-oriented programs. Our approach adapts the state-based testing technique for object-oriented programs into the context of AOP aspects. This allows the testing of program behavior to be carried out using the same technique for those classes not involving aspects (*i.e.*, un-weaved classes) and for those base classes interacting with aspects and, hence, facilitate the testing work for AOP programs.

The current testing approach also exists several limitations. In particular, the construction of state-based test model requires manual efforts, which can affect the uses and scalability of the proposed approach. This limitation can be partially alleviated if the testing focuses only on those mission-critical base classes and their associated aspects. This is quite nature since some base classes and aspects, such as logging, may not exist observable behavior that concerns the logic correctness of AOP programs. In addition, the manual efforts can be partially reduced with the assistance of automatic state-based testing tools [4, 11].

Moreover, the test tree presented in section 4 could grow rapidly if the number of states and transitions in AsOSD becomes large. This indicates that the algorithm for generating the test tree may suffer from an exponential increase in computational complexity. This is a generic problem when performing reachability analysis for large finite state systems. The method to ease this problem is beyond the scope of this paper and many existing techniques, such as symbolic representation and state-space pruning, to tackle the problem can be found in the literature [6]. Another practical way to alleviate this problem is to apply the proposed approach only to unit testing of AOP programs. In unit

testing of AOP program, only one base class is considered at a time and the size of the test tree can be small and limited.

The proposed approach addresses only the state-based behavior between the base classes and aspects introduced by *method call* join points. However, the interactions between base classes and aspects can also be triggered through other kind of join point, such as variable access. Nevertheless, since the *variable access* join point is typically located within a method of the base class, the aspect invocation through a *variable access* join point can be extended and viewed as an invocation from a method call. As a result, the proposed approach is still applicable to this type of join point.

There are still other testing concerns raised by AOP programs which have not yet been explored in the proposed approach. For instance, the precedence of advices can be affected by the inheritance and domination relationships between the aspects. This suggests that the static structure of AOP programs may need to be considered in order to obtain correct advice order. Moreover, the exception handling constructs can possibly change the execution flow of advices, which is not yet addressed in the proposed approach.

6. RELATED WORK

Although AOSD is gradually gaining widespread popularity, most of existing research in AOSD has focused on the analysis modeling, design, and implementation techniques. Testing for AOP programs still receives a little attention in the AOSD community. In this section, we briefly review several existing work in AOP program testing.

Alexander *et al*. [3] present a potential fault model and four potential sources of fault that may exist in the AOP programs. The model can help testers to understand how faults and failures occur in AOP programs. In addition, Alexander *et al*. propose several testing criteria associated with their fault model. Based on the types of faults, the testing criteria suggest different testing focuses for AOP programs.

Massicotte *et al*. [13] propose an AOP program testing approach. The approach analyzes the program behavior depicted in the UML collaboration diagram. The main idea of the approach is to derive all possible weaving combinations between base class and woven advices and then verify the results of weaving adaptively.

Naqvi *et al*. [14] provide a summary of researches on testing AOP programs. In particular, they elaborate the potential fault model proposed by Alexander *et al*. [3]. Naqvi *et al*. provide a helpful discussion and illustrate examples on each fault category based on the fault model. Moreover, they review the strength and weakness of several testing approaches for AOP programs in the literature including data flow based testing, state-based testing, and aspect flow based testing. In addition, the effectiveness of those testing approaches is evaluated according to each fault category and several research directions for testing AOP programs are provided.

Zhao [22, 23] presents a unit testing method for AOP programs using data-flow based testing technique. The method first models the control flow of pure class without aspect weaving. The additional control flow introduced by the woven advices is then added into the clustering class to represent the control flow modified by the aspects. Finally, the potential problems of the AOP programs are analyzed through the def-use

paths derived from the modified control flow graph. Furthermore, Zhao *et al*. [24] propose a regression test approach for AspectJ programs based on various types of control flow graphs. Based on the flow graphs, test cases can be selected from the original test suite for the modified AspectJ programs.

Xie *et al*. propose a framework called Aspectra [17, 18] that can generate test inputs for testing aspectual behavior automatically. The Aspectra framework is extended from their previous work called Rostra [16] which can produce effective unit testing inputs for woven classes of AspectJ programs by omitting potential redundant test cases. In addition, the Aspectra also proposes aspectual branch coverage and interaction coverage to facilitate the quality assess of generated tests and to guide developers for improving test coverage of the generated tests, respectively.

Zhou *et al*. [25] present a testing approach and an associated tool for testing AOP programs. The approach adapts an incremental testing strategy which starts from testing regular classes, then weaves each aspect to the regular classes one-by-one, and finally tests the woven result among the classes, few aspects, and all aspects. The incremental testing strategy is corresponding to unit testing, integration testing, and system testing for AOP programs. Through the approach, testers can find any unexpected results in the early stage of program development.

Xu *et al*. [19] propose a unit testing method for AOP programs by leveraging the state-based testing technique. Their method is based on a model called Aspectual State Model (ASM) which is adapted from the known FREE (Flattened Regular ExprEssion) state model [4]. The ASM can represent the state-based behavior of an object as well as possible behavior changes introduced by the woven advices. Based on the ASM, test cases can be derived to test a single class with multiple weaving aspects.

In addition, Xu *et al*. [20] present an incremental testing strategy for AOP programs. The main idea of the incremental testing strategy is to reuse the test of concrete base class for testing aspects according to the state change impacts on the base class caused by the aspects. In particular, an extended state model for aspect and weaving mechanism is presented. The state model can facilitate the specification of the impacts on the state-based behavior of base class. In addition, several rules have been proposed for reusing the base class tests.

Furthermore, Xu *et al*. describe a model-based approach for testing integration aspects [21]. They point out that an aspect integrating separated concerns can contain various faults. Thus, they present an aspect-oriented state model to specify such integration aspects. By composing the state models of aspects and classes, they are able to generate test cases from the state models to test the integration aspects.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a state-based testing approach for AOP programs. The approach considers the dynamic behavior changes introduced by different advices in multiple aspects. A test model is presented to abstract the state-based behavior of AOP program with the consideration of the interactions between the base classes and aspects. In particular, the test model consists of OSD, CWM, and AsOSD. The OSD is used to represent the state-based behavior of AOP program before aspect weaving. The CWM is

used to abstract the advice weaving sequences and to analyze the changes of state variables after advice weaving. The AsOSD is then used to combine the results of OSD and CWM and represent the state-based behavior of AOP programs after aspect weaving.

Based on the test model, a procedure is described to construct a test tree by analyzing the reachability of the set of state machines in the AsOSD. From the test tree, we can derive test cases, which are sequences of method calls of base classes, in order to uncover the state behavior errors in the AOP programs. In addition, an auto vending machine example is provided to show the notions of the test model. The effectiveness of the proposed approach is also illustrated by showing how can a behavioral bug in the vending machine example be detected through the testing approach.

The approach presented in this paper provides an attempt to ensure the correctness of dynamic behavior for AOP programs. There are still many areas required to be explored in order to assure high quality for AOP programs. In the future, we plan to extend the proposed approach to take into account the static structures of AOP programs. In specifically, we plan to explore various relationships among the aspects as well as the relationships between the base classes and aspects. We also plan to study how the different relationships can implicitly influence the weaving order of advices and affect the dynamic behavior of AOP programs.

Moreover, we plan to extend our approach to address different advice precedence rules and investigate how different types of pointcut in AspectJ may affect the weaving order of advices. In addition, we plan to develop a testing tool to automate the proposed approach so that the test model, such as CWM, and the test cases can be generated automatically. With the help of the tool, we also plan to conduct more experiments on real-world examples to elaborate the proposed test model and report the experimental results and experiences.

REFERENCES

1. Aop@work: Aop tools comparison, http://www-128.ibm.com/developerworks/java/library/j-aopwork1/.
2. The AspectJ 5 Development Kit Developer's Notebook, http://www.eclipse.org/aspectj/doc/released/adk15notebook.
3. R. T. Alexander, J. M. Bieman, and A. A. Andrews, *Towards the Systematic Testing of Aspect-Oriented Programs*, Technical Report No. CS-4-105, Department of Computer Science, Colorado State University, Fort Collins, CO, 2004.
4. R. V. Binder, *Testing Object-Oriented Systems: Models*, *Patterns*, *and Tools*, Addison-Wesley Professional, Boston, 1999.
5. M. R. Blaha and J. R. Rumbaugh, *Object-Oriented Modeling and Design with UML*, 2nd ed., Prentice Hall PTR, New Jersey, 2004.
6. E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*, MIT Press, 2000.
7. L. A. Clarke, "A system to generate test data and symbolically execute programs," *IEEE Transactions on Software Engineering*, Vol. SE-2, 1976, pp. 215-222.
8. A. Colyer, A. Clement, G. Harley, and M. Webster, *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools*, Addison-Wesley Professional, 2004.

9. R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, *Aspect-Oriented Software Development*, Addison-Wesley Professional, Boston, 2004.
10. W. L. Hursch and C. V. Lopes, *Separation of Concerns*, Technical Report No. NU-CCS-95-03, College of Computer Science, Northeastern University, Boston, MA, 1995.
11. D. C. Kung, P. Suchak, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen, "On object state testing," in *Proceedings of the 18th Annual International Computer Software and Applications Conference*, 1994, pp. 222-227.
12. C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*, 3rd ed., Prentice Hall, New Jersey, 2004.
13. P. Massicotte, M. Badri, and L. Badri, "Generating aspects-classes integration testing sequences: a collaboration diagram based strategy," in *Proceedings of the 3rd ACIS International Conference on Management and Applications*, 2005, pp. 30-37.
14. S. A. A. Naqvi, S. Ali, and M. U. Khan, "An evaluation of aspect oriented testing techniques," in *Proceedings of the IEEE Symposium on Emerging Technologies*, 2005, pp. 461-466.
15. A. J. Offutt and A. Abdurazik, "Generating tests from UML specifications," in *Proceedings of the 2nd International Conference on the Unified Modeling Language*, LNCS 1723 , 1999, pp. 416-429.
16. T. Xie, D. Marinov, and D. Notkin, "Rostra: a framework for detecting redundant object oriented unit tests," in *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, 2004, pp. 196-205.
17. T. Xie and J. Zhao, "A framework and tool supports for generating test inputs of AspectJ programs," in *Proceedings of the 5th International Conference on Aspect-Oriented Software Development*, 2006, pp. 190-201.
18. T. Xie, J. Zhao, D. Marinov, and D. Notkin, "Automated test generation for AspectJ programs," in *Proceedings of the AOSD Workshop on Testing Aspect-Oriented Programs*, 2005.
19. D. Xu, W. Xu, and K. Nygard, "A state-based approach to testing aspect-oriented programs," in *Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering*, 2005, pp. 366-371.
20. D. Xu and W. Xu, "State-based incremental testing of aspect-oriented programs," in *Proceedings of the 5th International Conference on Aspect-Oriented Software Development*, 2006, pp. 180-189.
21. W. Xu and D. Xu, "State-based testing of integration aspects," in *Proceedings of the 2nd Workshop on Testing Aspect-Oriented Programs*, 2006, pp. 7-14.
22. J. Zhao, "Data-flow-based unit testing of aspect-oriented programs," in *Proceedings of the 27th Annual International Computer Software and Applications Conference*, 2003, pp. 188-197.
23. J. Zhao, "Tool support for unit testing of aspect-oriented software," in *Proceedings of the OOPSLA Workshop on Tools for Aspect-Oriented Software Development*, 2002.
24. J. Zhao, T. Xie, and N. Li, "Towards regression test selection for AspectJ programs," in *Proceedings of the 2nd Workshop on Testing Aspect-Oriented Programs*, 2006, pp. 21-26.
25. Y. Zhou, D. Richardson, and H. Ziv, "Towards a practical approach to test aspect-

oriented software," in *Proceedings of the Workshop on Testing Component-Based Systems*, 2004.

Chien-Hung Liu (劉建宏) received his B.S. degree from the Electrical Engineering Department of National Cheng Kung University in 1989, his M.S. degree from the Electrical Engineering Department of University of Southern California in 1994, and his Ph.D. degree from the Computer Science and Engineering Department of the University of Texas at Arlington in 2002. He joined the Computer Science and Information Engineering Department at National Taipei University of Technology, Taiwan as assistant professor in 2003. His research interests include software testing and maintenance, web services, software processes, and embedded software.

Chuan-Wen Chuang (張傳文) received his M.S. degree in Information Management from Chinese Culture University in 2005. He is now a Ph.D. student in the Graduate Institute of Computer and Communication Engineering at National Taipei University of Technology, Taiwan. His research interests include software engineering, model driven architecture, and software testing.