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, Taiwan
E-mail: {cliu, s4419012}@ntut.edu.tw

ABSTRACT

In recent years, Aspect Oriented Programming (AOP) has become an emerging technology due to its ability of supporting separation of concerns. 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. In this paper, we propose a state-based testing approach for AOP programs. The approach considers the state behavior changes introduced by multiple aspects. A state-based test model is presented to abstract the behavior caused by various weaving sequences of aspects. Based on the model, test cases can be derived so as to uncover the state behavior errors of AOP programs.

1: INTRODUCTIONS

Aspect-oriented software development and AOP were first introduced at Xerox PARC in 1990s to empower the programming capabilities of conventional object-oriented programming, especially for the concept of separation of concerns (SoC) [7] [8]. The basic idea of SoC is to divide the program requirements into two primary categories – the functional and non-functional requirements. Ideally, with SoC, each crosscutting concern can be designed and implemented independently in order to enhance the extendibility and maintainability.

Existing object-oriented programming suffers from a serious limitation to adequately modularize crosscutting concerns in software development. The code related to those concerns is often duplicated within several classes. AOSD proposes a new abstraction dealing with separation of crosscutting concerns in software development. Through this technique, the code corresponding to the non-functional concerns can be factored out into standalone code modules called aspects. This new programming model enables better flexibility and manageability in software development.

However, the AOP paradigm also raises a new testing problem due to the potential changes of original program behavior after weaving the aspects. Therefore, it becomes a new challenge to make sure that the behavior of the woven program aligns to the program specifications.

Several methods have been proposed for testing AOP programs. Alexander et al. [3] present a potential fault model along with associated testing criteria for AOP testing. Zhao [14] proposes a data flow testing approach for AOP. The approach considers the data flow anomaly caused by two kinds of weaving: a single aspect to multiple classes and multiple aspects to a single class. Xu et al. [13] describe a state-based testing approach to test AOP programs. They present a model called Aspectual State Model (ASM), which is based on the known FREE (Flattened Regular ExprEssion) state model [4]. Their ASM can derive test cases for the scenario of a class weaved by multiple aspects. Massicotte et al. [11] present a testing strategy which can generate test sequences based on the dynamic interactions between aspects and classes from the UML collaboration diagram. Naqvi et al. [12] have also summarized some AOP testing researches and their testing techniques used.

In this paper, we present a state-based testing approach for AOP programs. The approach considers the state behavior changes introduced by multiple aspects. A crosscutting model is suggested to abstract different aspect weaving sequences. The state behavior of AOP program is then captured using aspect object state diagram (AsOSD) with consideration of the effects caused by aspect interactions. From the AsOSD, test cases can be generated to uncover the behavioral errors of AOP programs.

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, we present the proposed approach to model different weaving sequences and state behavior of AOP programs. In section 4, we describe how to generate test cases based on the proposed model. The last section summarizes our conclusions and future work.

2: A MOTIVATION EXAMPLE

Traditionally, the state behavior testing for object-oriented programs considers only the interactions between the classes. This may not be adequate for AOP programs since some behavioral errors can happen only in the interactions between the classes and aspects. We

will illustrate how such behavioral errors can occur through a simple example.

Consider a basic coin box prototype of an auto vending machine implemented using AspectJ, a popular Java-based AOP programming language [1] [6]. It accepts coins inserted by customers and drops a drink when the amount of coins received is enough. To simplify this example, the controls to the physical devices are omitted. The coin box keeps track of the coins inserted by a customer (denoted *curQtrs*) and the total coins received from all vending transactions (denoted *totalQtrs*), and 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
2) *ReturnQtr*(): Return all coins which inserted by a customer
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

Except the basic behavior, two additional control mechanisms of this vending machine are implemented in the *DrinkCheck* aspect (in the *DrinkCheck.aj*) and *VendControl* aspect (in the *VendControl.aj*) in order to provide the interaction controls of the coin box class. The *DrinkCheck* aspect provides the inventory checking control that checks the amount of available drinks of the vending machine. It will disable the functionality of *Vend*() method of the coin box class when all drinks in the storage of the vending machine are sold-out. The *VendControl* aspect provides the vending control. It enables the functionality of *Vend*() method of the coin box class when the amount of coins received is enough. The separations of *DrinkCheck* and *VendControl* aspects from the coin box class allow to avoid invasive composition between the base class and the aspects.

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 rule of advice ordering. Note that AOP can weave advices into crosscutting member functions (called *joint points*) by its native weaving rule in a multi-advice and multi-aspect AOP program. For example, AspectJ in a multi-advice aspect, the *before* and *around* advices get higher precedence than the *after* advices. In addition, for a multi-aspect AOP program, the weaving precedence will be random if no any precedence among the aspects are declared. In the vending machine example, a precedence declaration in *AspectInfrastructure.aj* enforces the advices in *DrinkCheck* aspect getting higher precedence than the advices in *VendControl* aspect because 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. The completed source code of this vending machine program is listed in Figure 1.

```
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*

```
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*

Figure 1. Source code of the vending machine example

In Figure 1, the coin box is implemented in a Java class as a core concern. The control mechanisms *DrinkCheck* and *VendControl* aspects 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 object 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 category and then integrates them together, the actual implementation may not restrictedly follow the rule of separation of concern if it is not clearly defined in the specification.

By carefully analyzing the source code of the vending machine example, one may find a behavioral 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 specification; 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 behavioral error 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 storage is out of stock. This error is introduced by an incorrect state transition and can be removed by changing the state of *allowVend* to false in the *after*():*OnVend*() advice after a successful purchase. The modified advice is shown in Figure 2.

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

Figure 2. The modified *after*():*OnVend*() advice

3: THE AOP TESTING APPROACH

In this section, we present an approach for testing AOP programs based on program specifications or source code. The approach consists of several steps in order to generate test cases for uncovering behavioral 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 behavior of OO programs can be represented using state-transition diagrams or UML statecharts (or other variants) [5] [10]. In this paper, we adapt the state behavior representation presented in [9] for AOP programs. Basically, in [9] the state 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 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 both the objects and aspects since the woven aspects become an extension of the objects. Thus, the data members (called *intertype declaration*) and the member functions (called *advice*) of aspects need to be considered in capturing the state behavior of the AOP program [2] [6]. Specifically, in our representation, the data members of objects and aspects involving state behavior are considered as state variables. However, for capturing 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 member function can only be invoked indirectly through the member function of an object. Most important, the member functions of an aspect cannot be invoked individually. Instead, they will be called automatically in a single invocation of the aspect according to the rule 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.

Consider the vending machine example in Figure 1. There are four member data *totalQtrs*, *curQtrs*, *drinkAvailable*, and *allowVend* in the class and aspects. By analyzing the example, 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 or ranges of these state variables are given in Table 1. In addition, 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.

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

State variables	Value domain
CCoinBox.curQtrs	$\leq 0, >0$ and $\leq 1, >1$
DrinkCheck.drinkAvailability	$>0, \leq 0$
VendControl.allowVend	*false, true*

a. *State Variables*

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

b. *State transitions obtained from CCoinBox*

3.2: THE OBJECT STATE DIAGRAM

Based on the state variables and transitions in Table 1, the Object State Diagram (OSD) [9] for the vending machine example can be constructed as shown in Figure 3. The OSD is a set of concurrent and communication state machines that can be used to represent the state behavior of an OO program. It abstracts the state behavior of each individual state variable. The overall behavior of the OO program is then represented by the interactions among the state machines of the state variables. This 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 behavior of an AOP program before the effects of aspect weaving are considered. The following steps outline the procedure of constructing an OSD from the state variables and transitions.

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 and draw an arrow to the initial state of each state variable in the diagram. These arrows indicate the state initialization of the variables.
3) For each state *S* identified in step 1, check each transition to see if the transition can cause a state change from *S* to another state *S'*. If yes, draw an arrow from *S* to *S'*; otherwise, draw an arrow form *S* back to itself. Label the arrow with an appropriated guard condition [5] and the name of the transition.

It should be noted that Figure 3 does not show the transitions for those states derived from aspect's member data. 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 aspect. We will show how to obtain the transitions for these states in sections 3.3 and 3.4.

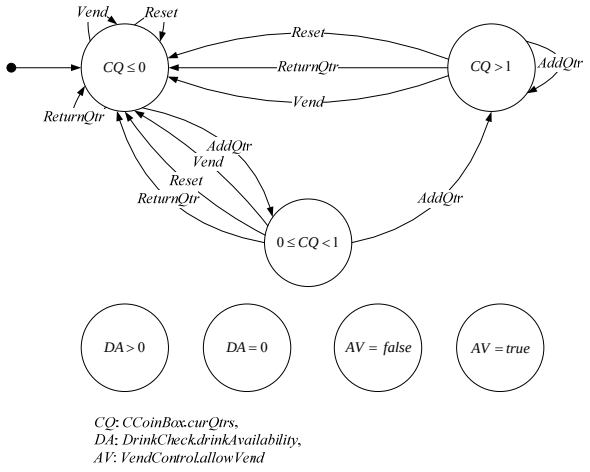


Figure 3. The object state diagram for the vending machine example

3.3: THE CROSSCUT WEAVING MODEL

As mentioned in section 3.1, the member functions (i.e., advices) of an aspect will be called internally and automatically when the joint 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 joint point; 2) the joint point followed by the *after* advice; or 3) the *around* advice. Note that the *around* advice can invoke the joint point during its execution depending on whether the function *proceed*() is called or not. For a multi-advice woven scenario, the order of the advices in general 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].

Since the invocation of the joint point can cause a series *before*, *after*, or *around* advices to be executed, the original execution result of the joint 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* 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 sequence of advices and joint point is first examined based on the weaving rules. The execution path of each advice is then depicted one by one in order to obtain possible values of state variables after integrating the advices and joint point. Notice that the *around* advice may or may not invoke the joint point. In such a case, the execution path of the *around* advice need to be analyzed and the joint point will be added into the path according to the logic of the advice.

Figure 4 shows the crosscut weaving model for the vending machine example. In the figure, there are four joint points. Each joint point is connected to proper advices according to the weaving rules. The execution paths of each advice are depicted and the values of state

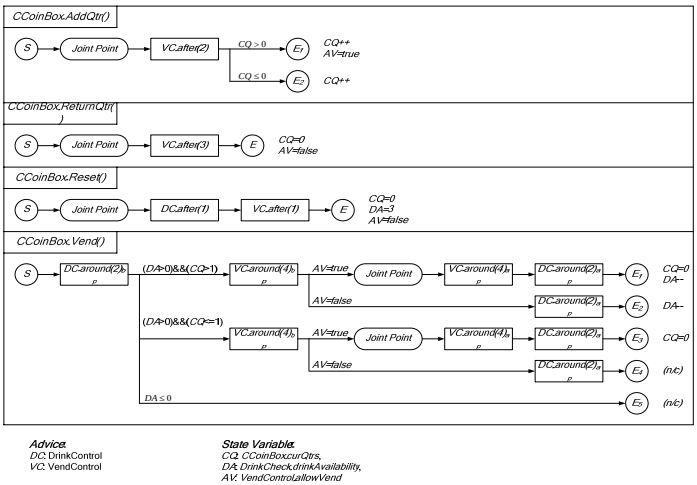


Figure 4. The crosscut weaving model for the vending machine example

variables after the paths being executed are also denoted. For example, the joint 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 *curOtrs* and *allowVend* will be changed (denoted *CQ* and *AV* in the figure).

Moreover, the joint 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 joint point *CCoinBox.Vend*() will be added into *VC.around(4)* according to the multi-advice ordering rule [2]. 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 outline the procedure of constructing the crosscut weaving model for a joint point:

1) Create a start point (denoted as a circle-s) in the diagram. The start point represents the start of the execution path for the joint point.
2) Based on the aspect ordering rule, add the first member function (can be a *before* advice, an *around* advice, or a joint point) that can be invoked into the path. Denote this new added advice or joint point as *C*. Add an arrow from the start point to *C*.
3) Following the aspect ordering rule and the control structure of *C*, add the next member function being invoked right after *C* into the path. If a branch structure exists in *C*, add the member function into a proper branch path. Add an arrow from *C* point to this new added advice or joint point.
4) If all the paths in *C* have been connected to a member function or to the end point, select one member function connected by *C*. Let *C* be the new selected function.
5) Go to step 3 and add member function that can be invoked right after *C* into the path till no more function can be invoked.
6) At each end of the execution paths for the joint point, add an end point (denoted as a circle-e) and label the values of state variables after executing the path.

3.4: ASPECT OBJECT STATE DIAGRAM

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

Basically the AsOSD can be constructed in three steps by analyzing the crosscut weaving model and the OSD before the aspect weaving. The first step is to identify the transitions for aspect state variables. From each execution path of the joint point in the crosscut weaving model, the aspect member functions (i.e., transitions) and their results can be easily identified. For instance, consider the execution path of the *CCoinBox.AddOtr* joint point in Figure 4. There is a transition *VC.after(2)* that will change the state variable *AV* to *true*. Figure 5 shows the transitions of aspect state variables for the vending machine example obtained from the crosscut weaving model.

The second step is to identify the transitions of object state variables that are influenced by aspect weaving. From the crosscut weaving model, we can find that the original four transitions *CCoinBox.AddOtr()*, *CCoinBox.Return()*, *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 Figure 4, there are two execution paths for the *CCoinBox.AddOtr()* function. Each path has different guard condition and result. This means that, after weaving, the original transition *CCoinBox.AddOtr()* will turn into two variants that can cause effects to the state variables different from the original function *CCoinBox.AddOtr()*. Note that in Figure 4 there are three paths in which the joint point is not invoked. This indicates that the weaving will cause the effects of the original transitions to be ignored.

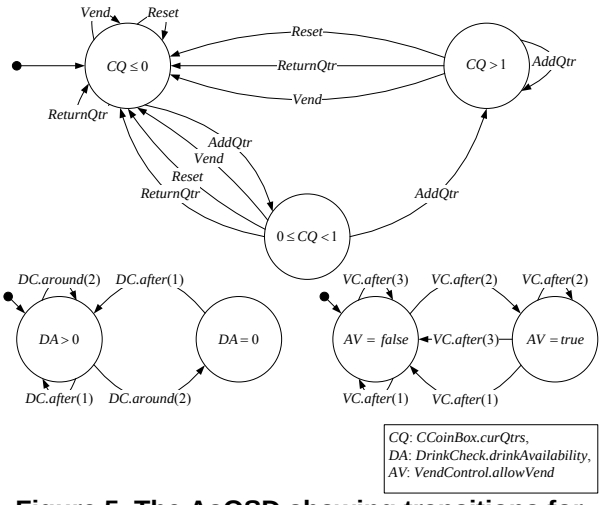


Figure 5. The AsOSD showing transitions for aspect state variables

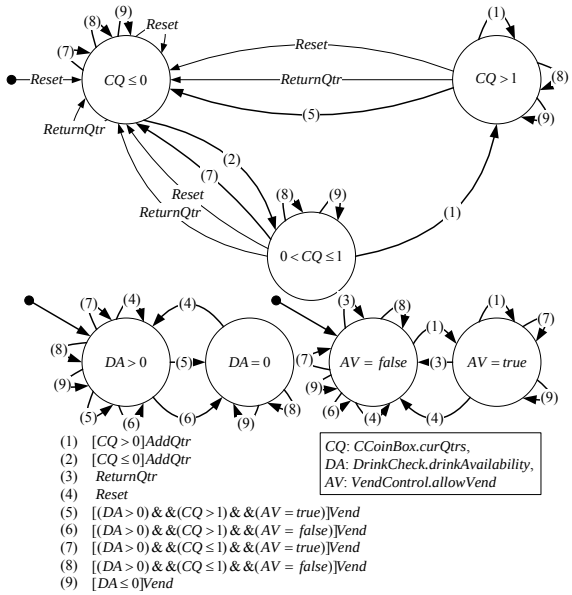
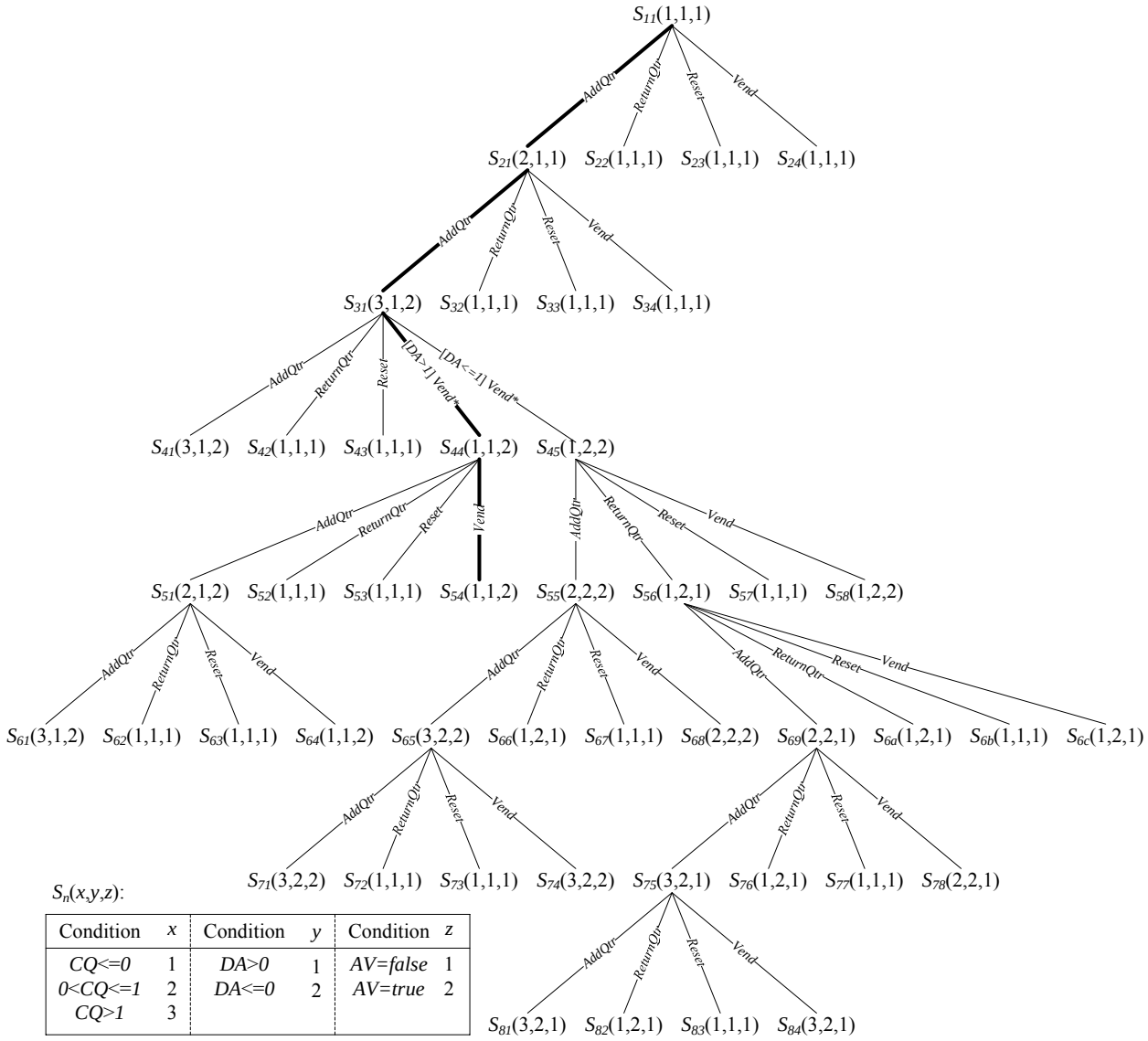


Figure 6. The aspect object state diagram for the vending machine example

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 transition with a proper transition variant via analyzing the corresponding state value and guard condition of the variant. For the transitions of aspect state variables, the original transition can be replaced by the variant that invokes the transition. For example, the transition *VC.after(2)* can be replaced by [*CQ>0*]*AddOtr()* since *VC.after(2)* is triggered by *AddOtr()*under the condition *CQ>0*. Figure 6 shows the AsOSD obtained from Figure 5 by replacing the original transitions with proper transition variants.

4: TEST CASE GENERATION

Based on the AsOSD, a test tree for AOP programs can be constructed. The test tree is presented in a spanning tree and consists of a number of nodes and links [9]. Each node in the test tree represents a state of the AOP program, and each link represents a state transition



$S_n(x,y,z)$:

Condition	x	Condition	y	Condition	z
CQ<=0	1	*DA>0*	1	*AV=false*	1
0<CQ<=1	2	*DA<=0*	2	*AV=true*	2
CQ>1	3				

Figure 7. The test tree for the vending machine example

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 parentheses is the index of value domain of corresponding state variable. The following steps briefly outline the procedure of constructing the test tree for AOP programs:

1) Based on the AsOSD, place an initial state as the root of the test tree. The initial state can be identified as the composition of the initial state of each state variable. In the vending machine example, the initial state, denoted $S_{1(CQ\leq 0,\ DA>0,\ AV=false)}$, represents that the state is the first state node in the test tree and the values of the composite state variables are $CQ \leq 0$, $DA>0$, and $AV=false$. Let root be the source node.
2) Let $S_{n(i, j,..., k)}$ be the source node under examined. If there is a transition $[c]t$ that can leads state *j* to *j'*, and state *j* satisfies condition *c*, then creates a target node $S_{n+1(i, j',..., k)}$ and links the source node to the target node.
3) If the composite state (*i*, *j'*,…,*k*) of the target node has already occurred at a higher level in the tree, the target node becomes a leaf node of the tree. If not, let the target node be the source node.
4) Repeat step 2 in order to derive all possible state transitions till all target nodes become leaf nodes.

Figure 7 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 child node of the test tree. The test case represents a sequence of member function calls and can be useful for detecting the behavioral errors of AOP programs. For example, from the test tree, a behavioral error of 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 by an external client to the CCoinBox object: *insert a coin → insert a coin → push the vend button on the vending machine and get a drink → push the vend button again and get another free drink without inserting any additional coin*.

5: CONCLUSION AND FUTURE WORK

In this paper, we have presented a state-based testing approach for AOP programs. The purposed approach allows deriving test cases for uncover the behavior errors of AOP programs. In particular, a crosscut weaving model and an AsOSD are presented to capture the state behavior changes introduced by multi-aspect weaving. An example is provided to illustrate effectiveness of the proposed approach.

In the future, we plan to extend the proposed approach to handle inheritance relationship and to investigate other weaving rules of AOP programs. In addition, a behavioral testing framework for AOP programs is in progress so as to facilitate test automation for AOP programs.

REFERENCES

[1] Aop@work: Aop tools comparison. Available at http://www-128.ibm.com/developerworks/java/library/j-aopwork1/.

[2] The AspectJ 5 Development Kit Developer's Notebook. Available at 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, Department of Computer Science, Colorado State University, Fort Collins, CO.

[4] R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. The Addison-Wesley Object Technology Series. Addison-Wesley Professional, Boston, 1st edition, October 1999.

[5] M. R. Blaha and J. R. Rumbaugh. *Object-Oriented Modeling and Design with UML*. Prentice Hall PTR, New Jersey, 2nd edition, November 2004.

[6] 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, December 2004.

[7] R. E. Filman, T. Elrad, S. Clarke, and M. Akşit. *Aspect-Oriented Software Development*. Addison-Wesley Professional, Boston, October 2004.

[8] W. L. Hursch and C. V. Lopes. Separation of concerns. Technical Report NU-CCS-95-03, College of Computer Science, Northeastern University, Boston, MA, February 1995.

[9] D. C. Kung, P. Suchak, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen. On object state testing. *In Proceedings of the Eighteenth Annual International Computer Software and Applications Conference (COMPSAC'94)*, pages 222–227, November 1994.

[10] C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice Hall PTR, New Jersey, 3rd edition, October 2004.

[11] P. Massicotte, M. Badri, and L. Badri. Generating aspects-classes integration testing sequences: A collaboration diagram based strategy. In *Proceedings of the third ACIS International Conference on Management and Applications*, pages 30–37, August 2005.

[12] 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*, pages 461–466, September 2005.

[13] D. Xu, W. Xu, and K. Nygard. A state-based approach to testing aspectoriented programs. In *Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering (SEKE'05)*, July 2005.

[14] J. Zhao. Data-flow-based unit testing of aspect-oriented programs. In *Proceedings of the 27th Annual International Computer Software and Applications Conference, 2003. COMPSAC 2003*, pages 188–197, November 2003.