International Journal of Computer Applications (0975 – 8887)
Volume 33– No.8, November 2011

Testing Aspect-Oriented Programs with UML Activity Diagrams

Somayeh Madadpour
Department of Computer Engineering
Science and Research Branch, Islamic Azad University
Tehran, Iran

Seyed-Hassan Mirian-Hosseinabadi
Department of Computer Engineering
Sharif University of Technology
Tehran, Iran

Vahdat Abdelzad
Department of Computer Engineering
Science and Research Branch, Islamic Azad University
Tehran, Iran

ABSTRACT
Aspect-Oriented Programming is a software engineering paradigm that offers new constructs, such as join points, pointcuts, advices, and aspects in order to improve separation of crosscutting concerns. The new constructs bring new types of programming faults with respect to crosscutting concerns, such as incorrect pointcuts, advice, or aspect precedence. In fact, existing object-oriented testing techniques are not adequate for testing aspect-oriented programs. As a result, new testing techniques must be developed. In this paper, an approach based upon UML activity diagram for testing aspect-oriented programs is presented. The proposed approach focuses on integration of one or several crosscutting concerns to a primary concern and tests whether or not an aspect-oriented program conforms to its expected crosscutting behaviors. The proposed approach generates test sequences based on interaction between aspects and primary models, and verifies the execution of the selected sequences. It also, follows an iterative process which causes to discover faults easily and quickly. The approach is based on several test criteria that we defined. To illustrate the approach, we use a case study which its results show that the approach is capable of revealing several aspect-specific faults.

General Terms
Verification, Modeling

Keywords
Aspect-Oriented Programming, Model-Based Testing, Aspect-Oriented Modeling, UML Activity Diagrams, Test Sequences

1. INTRODUCTION
Aspect-Oriented Programming (AOP) was first introduced at Xerox PARC in 1997s to improve the programming capabilities of conventional object-oriented programming, especially to support the principle of Separation of Concerns (SoC) in software development [1, 2]. Typically, a concern can be customer required property or a technical interest, such as security, that can spans the entire system. The central idea of SoC is to modularize the crosscutting concerns of a system, such as synchronization, memory management, and persistency in order to enhance the reusability, extendibility, and maintainability.

Traditional programming languages such as, procedural and Object-Oriented (OO) 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 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 and allow developers to separate crosscutting concerns that scatter across different procedures (or classes) of a system into modular units called aspects.

AspectJ, as a widely-used AOP programming language for java, introduces several new programming constructs, such as join point, pointcut, advice, and aspect [3]. 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 that are difficult to express cleanly using traditional programming languages, such as non-functional requirements, can be factored out into aspects in order to achieve the principle of SoC.

One of the capabilities of the aspect-oriented programming languages is facilitating the defining, specifying, designing, and constructing aspects and enforcing a better coding style. However, some errors that generated by the undisciplined programmers or by the misunderstanding of requirements in the system during development cannot be prevented. The new programming constructs and their interactions presented in the aspect-oriented programming languages are necessary to be tested. Most important, in AOP paradigm because of weaving the aspects into the original programs, the behavior of the system may be changed. Therefore, it becomes a testing challenge to make sure whether or not the behavior of the woven AOP program conforms to its program specifications.

To reveal aspect-specific faults, we are motivated to inspect model-based testing, i.e. testing whether or not aspect-oriented programs and their primary concerns are in accordance with their corresponding behavior models. Model-based testing is attractive because of several benefits [4, 5], including:

- The modeling activity in the testing process can help to explain the requirements and to improve the relationships between developers and testers.
- If design models are available, can be reused to testing purposes.

- The model based testing process can be (partially) automated.
- Fault detection capability can be improved by model-based testing as well as testing cost can be reduced by automatically generating and executing test cases.

Pretschner et al. [5, 6] illustrated that, for the case study of an automotive network controller, a six-fold increase in the number of model-based tests has led to an 11% increase in detected errors. Dalah et al. [7] demonstrated that with applying model-based testing on four large scale case studies, generated test cases revealed numerous defects that were not exposed by traditional approaches. Blackburn et al. [8] by applying model-based testing methods and tools on the Mars Polar Lander (MPL) software were able to identify its error that is believed to have caused the MPL to crash to the Mars surface on December 3, 1999.

Here we propose an UML activity diagram [9] based approach to testing whether or not an aspect-oriented program conforms to its expected crosscutting behavior. The approach focuses on the problem related to weaving one or several crosscutting concerns to a primary concern. The approach follows an iterative process. It consists of generating, in a first step, test sequences corresponding to different scenarios of the activity diagram of the primary concern under test. This is done to reduce the complexity of the testing process and to remove the likely faults related to the primary concern. In a second step, crosscutting concerns are integrated into primary concern in an incremental way and then, test sequences from the integrated model are generated. The primary objective is to verify that the original behavior of the primary concern is not changed by aspects, and to ensure that aspects behave correctly. Verification process of the selected sequences (test sequences generated in two previous steps) is done in the third step.

In addition, such an incremental approach to testing aspect-oriented programs can reduce the complexity of detecting eventual conflicts between aspects. We focus, in the context of our approach, on the conflicts that appear in the integration of one or several aspects to a primary concern. We define some test criteria related to the new dimensions introduced by the integration of aspects to the primary concern. For realization of our proposed approach we focus on AspectJ, however, our approach is general and may be adapted to other aspect implementations.

The rest of the paper is organized as follows: in Section 2, we present a survey of related works. Section 3 is an overview of our approach to testing aspect-oriented programs. In section 4, we discuss our approach in detail in four sub sections. Section 5 illustrates our approach using a case study. Finally, Section 6 gives a general conclusion and some future work directions.

2. RELATED WORK

While AOP provides a greater flexibility for modularizing crosscutting concerns, it cannot provide correctness by itself and raises new challenges for testing aspect-oriented programs. Alexander et al. [10] have proposed a fault model for aspect-oriented programming, including six types of aspect 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. We believe that, while this fault model has not yet constituted a fully developed testing approach, it is certainly useful for developing testing tools and strategies for aspect-oriented programs.

Zhao in [11] has proposed a data flow-based unit testing approach for aspect-oriented programs. For each aspect or class, the approach performs three levels of testing, i.e., intra-module, inter-module, and intra-aspect/ intra-class testing. Definition-Use (DU) pairs are calculated to determine what interactions between aspects and classes must be tested. Zhao and Rinard [12] have also exploited system dependence graphs to capture the additional structures present in many aspect-oriented features such as join points, advice, aspects, and various types of interactions between aspects and classes. Control flow graphs are constructed at both module and system level, and code based test suites are derived from control flow graphs. Zhou et al. [13] propose a unit testing strategy for aspects. Their approach is presented in four phases. The first step consists in testing classes to remove errors that are not in relation with aspects. Each aspect is integrated and tested separately in a second step. During the third step, all aspects are integrated and tested in an incremental way. Finally, the system is entirely retested. This approach is based on the source code of the program under test. Xie et al. [14, 15] propose a framework called Aspectra to automatically generate test inputs for AspectJ programs, where a wrapper class is created for each base class under test. The above works concentrate on code based testing. They address the question of "how much is the program being covered by testing?" other than "does the program satisfy the requirements?". In comparison, our approach focuses on testing whether or not aspect-oriented programs conform to aspect-oriented design models.

Xu et al. proposed different approaches for testing aspect-oriented programs [16, 17, 18]. They proposed in [16] a state-based approach for unit testing aspect-oriented programs. Their approach is based on a model called Aspectual State Model (ASM) that is an extension to the known FREE (Flattened Regular ExprEssion) state model [19]. The ASM represent the state-based behavior of an object and also possible behavior changes introduced by the woven advices. Once the ASM is created, it can be transformed into a transition tree, which implies a test suite for adequately testing object behavior and interaction between classes and aspects in terms of message sequences. In [17], they presented an incremental testing approach for aspect-oriented programs. The main idea of this approach is to reuse the base class tests for testing aspects according to the state-based impact of aspects on their base classes. In particular, an extended state model 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 is presented. In addition, several rules have been proposed for maximizing reuse of concrete base class tests for aspects. They also proposed in [18] a state-based approach for testing integration aspects. They indicate that an aspect integrating separated concerns, like other aspects, can contain various programming faults. Thus, they exploit an aspect-oriented state model to specify integration aspects. By composing the state models of aspects and classes, they are able to generate test cases for integration aspects from their state models. In addition, Xu et al. proposed in [20, 21] an approach based on different UML design models (class diagrams, aspect diagrams and sequence diagrams) to derive test cases covering the interactions between aspects and classes. Liu and Chang in [22] proposed a state-based testing approach for AOP programs. The approach considers the state-based behavior changes introduced by different advices in multiple aspects. A test model is suggested to depict the state based behavior of aspect-oriented program after aspect weaving. Based on this model, test cases can be derived in order to uncover the potential state behavior errors in the AOP programs. Badri et al. [23] presented a state-based unit testing technique for aspect-oriented programs and associated tool that focuses on the integration of one or several aspects to a class. It supports both the generation and verification of test sequences and its objective is to ensure that the integration is done correctly, without altering the original behavior of the classes. The above works focus on the behavior of a class where one or more aspects are weaved. Our research is related to the integration of one or more aspects to the behavior of a group of objects. We propose an UML activity diagram based approach to testing aspect-oriented programs that is capable reveal some of aspect-specific faults in the early stage of program development. Our work is based on a paper presented by Cui et al. [24] on modeling and integrating aspects with UML activity diagram. We improve this work from the perspective of model-based test sequences generation, and test sequences execution and verification.

3. OVERVIEW OF OUR APPROCH

The proposed approach consists of three main phases. The first phase is related to building activity model of the primary concern and generating the corresponding basic test sequences without integrating the aspects. The main goal of this step is to reduce the complexity of the testing process and to eliminate the faults that are not related to the aspects. The second phase is related to building aspect models, integrating them into the primary model incrementally, and generating the corresponding test sequences based on the testing criteria defined in Section 4.2. The main goals of this step are:

- To verify that the single aspect under test behaves correctly.
- To test and verify the interaction among aspects and to eliminate the errors that result from the presence of multiple aspects.

The third phase consists of verifying the execution of the selected sequences (the sequences which are generated in two previous phases). This process is supported by instrumenting the source code of the program under test. The major steps of our method are described in the following:

1. Building activity model of the primary concern and generating the basic test sequences.
2. Testing the primary concern separately.
3. Integrating an aspect. As long as there are aspects which are not integrated
    a. Building aspect model and weave it into primary model.
    b. Generating the test sequences affected or created by the aspect.
    c. Testing the primary concern with the integrated aspect.
    d. If there is no problem encountered, return to step 3.
4. Testing entirely the primary concern including aspects.
5. End.

To instrument the software under the test, we do use an aspect to capture a trace of the executed methods in a sequence. To test a sequence, we compile the program along with this aspect.

4. TESTING PROCESS: AN ITERATIVE APPROCH

In this section, we first discuss the aspect-oriented modeling with activity diagrams and integrating aspects with primary models, then we present the proposed testing criteria and describe test sequences generation process, and finally we present the test execution and verification process.

4.1 Aspect-Oriented Activity Diagram

Aspect-oriented activity diagrams motivated to capture the essential features (join points, pointcuts, etc.) of aspect-orientation for system modeling. We use of Cui et al. [24] work for modeling aspect-oriented programs with UML activity diagrams. Similar to the AOP notions, an aspect-oriented activity model consists of primary models, aspect models, and aspect precedence. Crosscutting concerns are depicted by aspect models, which consist of pointcut models and corresponding advice models. Both of last models specified by extended activity diagrams.

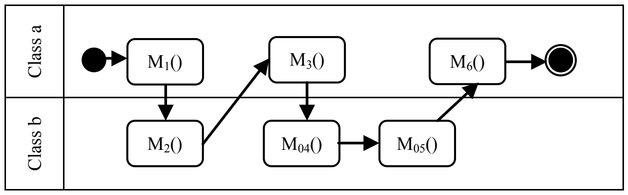
A pointcut model serves as a predicate to select join points (i.e. Nodes, Edges, and groups) from primary models and specifies advice model to be applied to these join points picked out from the primary models. An advice model specifies additional enhancements or constraints with respect to the crosscutting concern under study.

Crosscutting concerns are either sequential or parallel aspects that are running sequentially or in parallel with primary concerns. Sequential aspects are critical features that their running results determine the residual processes in primary models. Parallel aspects are uncritical and time consuming features that their running results should not influence the residual processes in primary models.

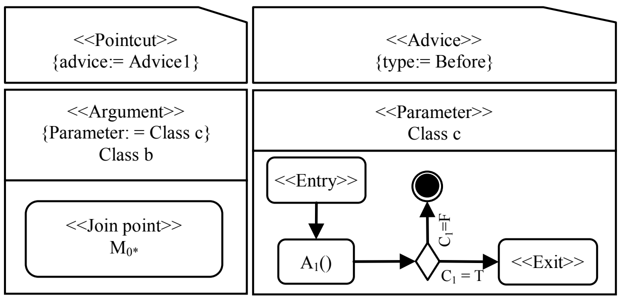
For illustrative purpose, Figure 1 shows a simple aspect-oriented activity model, including primary model, sequential aspect A_1 and parallel aspect A_2.

The aspect A_1 in Figure 1(b) consists of pointcut model Pointcut1 and the advice model Advice1 for Pointcut1.The pointcut model Pointcut1 depicted in Figure 1(b) (i) constructed to select the elements in primary models to which the sequential advice $A_1()$ will be applied. The pointcut model is stereotyped with <<Pointcut>>. A tagged value “advice” indicates the corresponding advice model is “Advice1”. This pointcut model describes the constraints of target join points from the three facets: the join point is an ActionNode, the name of node is $M_{04}()$ or $M_{05}()$; the Node belongs to an ActivityPartition named “Class b”; the predecessor and successor elements of the node are arbitrary. There is an argument element “Class b” in the pointcut model with tagged value “Parameter: =Class c”. This tagged value maps this argument to the formal parameter element “Class c” in Advice1. Figure 1(b) (ii) models the $A_1()$ concern as sequential advice which means that the $A_1()$ action needs to be performed before the join point nodes. The advice model is stereotyped with <<Advice>>. The tagged value “type”, which is tagged on <<Advice>>, indicates the type of the advice is “Before”. In the advice model, there is an element named “Class c” stereotyped <<Parameter>> that serve as a formal parameter. The two nodes stereotyped <<Entry>> and <<Exit>> denotes where the tokens will flow in from and flow out to the primary models respectively.

The aspect A_2 in Figure 1(c) consists of pointcut model Pointcut2 and the advice model Advice2 for Pointcut2. The pointcut model Pointcut2 depicted in Figure 1(c) (i) constructed to select the elements in primary models to which the parallel advice $A_2()$ will be applied. The join points should meet the following constraints defined in this pointcut model: the join point is an edge of ControlFlow; the edge has a predecessor node and FlowFinal successor node (the predecessor node is in an ActivityPartition named Class b, the name of the predecessor node should be “$M_{05}()$”). Figure 1(c) (ii) models the $A_2()$ concern as parallel advice. In the advice model, the “$A_2()$” action is fired at the join point and running in parallel with the residual flow of the primary model.

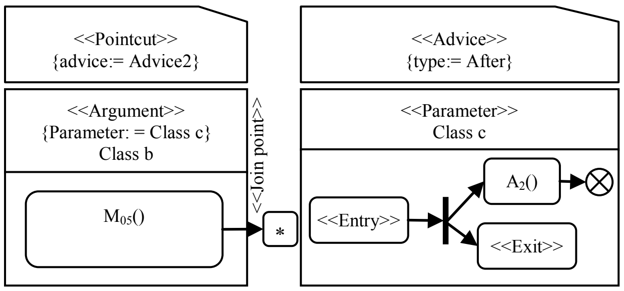


(a) The primary model



(i) Pointcut1 **(ii) Advice1**

(b) The A_1 aspect



(i) Pointcut2 **(ii) Advice2**

(c) The A_2 aspect

Figure 1: A simple aspect-oriented activity model

The semantics of an aspect-oriented activity model essentially depend on the weaving mechanism that composes aspect models into primary models. The result of composition is an integrated model. The integration is done by finding join points in primary models, initializing advice models, and weaving advices into primary models.

Figure 2 is the integrated model after weaving the Advice1 with the primary model. In this model, Advice1 was inserted before “$M_{04}()$” and “$M_{05}()$” nodes. Figure 3 is the integrated model after weaving the Advice1 and Advice2 with the primary model. In this model, Advice1 was inserted before the “$M_{04}()$” and “$M_{05}()$” nodes, and Advice2 was inserted after outgoing edge of the “$M_{05}()$” node. The definitions of aspect-oriented activity models and the weaving algorithm can be found in [24].

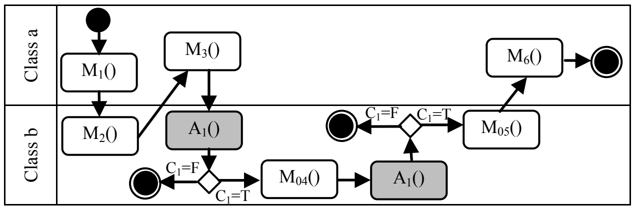


Figure 2: The integrated model with one aspect

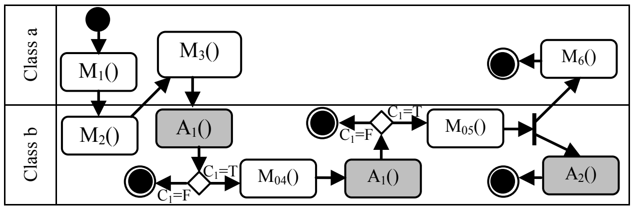


Figure 3: The integrated model with two aspects

4.2 Testing Criteria

A testing criterion is a rule or a collection of rules that impose conditions on testing strategies [25, 26]. It also can be used to evaluate a set of test cases (known as a test suite), or they can be used to guide the generation of test cases [25]. Testing criteria are used to determine what should be tested without telling how to test it. Testing engineers use those criteria to measure the quality of a test suite in terms of percentage [27]. In this section, we present several testing criteria.

The first criterion supports the generation of test sequences from classic activity diagrams of primary concerns. As mentioned previously, aspects have the capacity of affecting the behavior of primary concerns. We extend this criterion to cover new dimensions introduced by aspects.

4.2.1 Action Path Coverage Criterion

First, we consider a priority relation as given below.

Definition1: A priority relation, denoted as ‘<’, over a set of actions S_A in an activity diagram is defined as follows.

1. If an action $A_i \in S_A$ precedes a fork and $A_j \in S_A$ is the first action that exist in any thread originated from the fork, then $A_i < A_j$.
2. If an action $A_j \in S_A$ follows next to a join and $A_k \in S_A$ is the last action in any thread joining with the join, then $A_k < A_j$.
3. If $A_i \in S_A$ and $A_j \in S_A$ are two consecutive concurrent actions in a thread originated from a fork where A_i exist before A_j in the thread, then $A_i < A_j$.
4. If $A_i \in S_A$ and $A_j \in S_A$ are two consecutive non-concurrent actions in an activity diagram where A_i exist before A_j, then $A_i < A_j$.

An action path is a sequence of actions in an activity diagram, where each action in the path has at most one occurrence except those actions that exist within a loop. Note that an action path considers each output branch of the decision node, a loop at most two times, and choose one representative action path from a set of action paths that have same set of actions and satisfy same set of priority relations.

Next, we define the action path coverage criterion as follows: Given a set of action paths P_A for an activity diagram and a set of test cases T, for each action path $p_i \in P_A$ there must be a test case $t \in T$ such that when system is executed with a test case t, p_i is exercised.

The above criterion is related to classic activity diagrams. It is not cover aspects dependencies. Thus, we need to develop new criteria. The following criteria cover the new dimensions introduced by aspects. They are based on the faults model presented by Alexander et al. in [10].

4.2.2 Modified Action Path Coverage Criteria
All action paths that effected by one or several aspects must be re-tested.

4.2.3 Multi-Aspects Integration Coverage Criterion
If an action in primary model is affected by several aspects, the action paths that include that action must be re-tested at least once.

4.3 Test Sequences Generation
We directly generate test sequences from activity diagrams by following the testing criteria defined in section 4.2. We start by generating the basic test sequences corresponding to the primary model and testing the primary concern separately. This is done to reduce the complexity of the testing process and to remove the faults related to the primary concern. Table 1 shows the generated basic test sequences from primary model depicted in Figure 1 (a).

Table 1. Generated basic test sequences

No	Test sequences
1	$M_1() \rightarrow M_2() \rightarrow M_3() \rightarrow M_{04}() \rightarrow M_{05}() \rightarrow M_6()$

When all basic test sequences regarding the primary concern are generated and tested, we proceed to aspects integration. Aspects are integrated in an incremental way, as mentioned previously, to facilitate errors detection. The precedence in which aspect models are integrated explicitly specified. The proceeding order to introduce advice does not have importance. According to the criteria established in Section 4.2, we generate the affected sequences by the aspects. These sequences will be re-tested. Table 2 shows the generated test sequences from simple-integrated model in Figure 2, and Table 3 shows the generated test sequences from multi-integrated model in Figure 3.

Table 2. Generated test sequences for simple-integrated model

No	Test sequences
1	$M_1() \rightarrow M_2() \rightarrow M_3() \rightarrow A_1()$
2	$M_1() \rightarrow M_2() \rightarrow M_3() \rightarrow A_1() \rightarrow M_{04}() \rightarrow A_1()$
3	$M_1() \rightarrow M_2() \rightarrow M_3() \rightarrow A_1() \rightarrow M_{04}() \rightarrow A_1() \rightarrow M_{05}() \rightarrow M_6()$
4	$M_1() \rightarrow M_2() \rightarrow M_3() \rightarrow M_{04}() \rightarrow M_{05}() \rightarrow A_2() \rightarrow M_6()$

Table 3. Generated test sequences for multi-integrated model

No	Test sequences
1	$M_1() \rightarrow M_2() \rightarrow M_3() \rightarrow A_1() \rightarrow M_{04}() \rightarrow A_1() \rightarrow M_{05}() \rightarrow A_2() \rightarrow M_6()$

4.4 Testing Process
The aim of testing process is basically to verify if the executed sequences are in accordance with the selected ones in one hand, and if obtained results are in accordance with the expected ones in other hand. We present the main phases of the testing process as follows.
For each generated sequence S_i:
1. Instrumenting the program under test.
2. Executing the program under test.
3. Analyzing the results.

4.4.1 Instrumenting the Program under Test
When all sequences are generated, we can start the testing process. In contrast with traditional instrumentation techniques, we do use aspects to capture a trace of the executed methods in a given sequence. The advantage of this approach is that we don't modify in any way the original source code of the program under test. Generally, in traditional instrumentation techniques, many lines of source code are introduced in the program under test. Those fragments of code may introduce unintentionally faults [28]. We generate an aspect to capture a trace of the executed methods in a sequence. When we want to test a specific sequence, we compile the program with the corresponding aspect. When a method involved in a sequence is executed, the tracking aspect will keep information about that execution.

4.4.2 Executing the Program under Test
We can execute the program under test, after completing the instrumentation phase. It mainly consists of running the program and testing a specific sequence. Tester is responsible for providing test data to ensure the execution of the selected sequences.

4.4.3 Analyzing Results
When a sequence has been successfully executed, we compare the executed methods with the expected ones. Because of the existence concurrent actions in an activity diagram, the expected method sequences may not be equal to executed method sequences. To solve this problem we consider the following definition:
Definition 2: let A be a sequence of executed methods, and B be a sequence of expected methods, A match B if A and B have same set of methods and satisfy same set of priority relations (defined in section 4.2).

Our approach is capable to discover four types of aspect-specific faults, including incorrect advice type, weak or strong pointcut strength, and incorrect aspect precedence. A fault of an incorrect advice type refers to using a type of advice different from the one defined in the design (for example, an after type may be used instead of a before type). A weak (or strong) pointcut means the implementation picks out extra (or misses expected) join points.

5. CASE STUDY

We have applied the above approach to the testing an AspectJ application (telecom) taken on AspectJ web site [29]. This example illustrates some ways that dependent concerns can be encoded with aspects. It uses an example of system comprising a simple model of phone connections to which timing and billing features are added using aspects, where the billing feature depends upon the timing feature. The classes of the system are:

- Customer that has name and area code fields and models customers.
- Connection (which is abstract) and two concrete classes Local and LongDistance, that model the physical details of establishing local and long distance connections between customers.
- Call that models telephone calls.
- Timer that models timers.

The aspects of the system are:

- Timing that implements the timing concern and measures the total connection time for each customer by starting and stopping a timer associated with each connection.
- Billing that implements the billing concern on top of timing concern and declares a payer to each connection and also makes sure that local and long distance calls are charged accordingly.
- TimerLog that implements a log to print the times when the timer starts and stops.

Figure 4 shows integrated model with two aspects Timing and Billing for the telecom example.

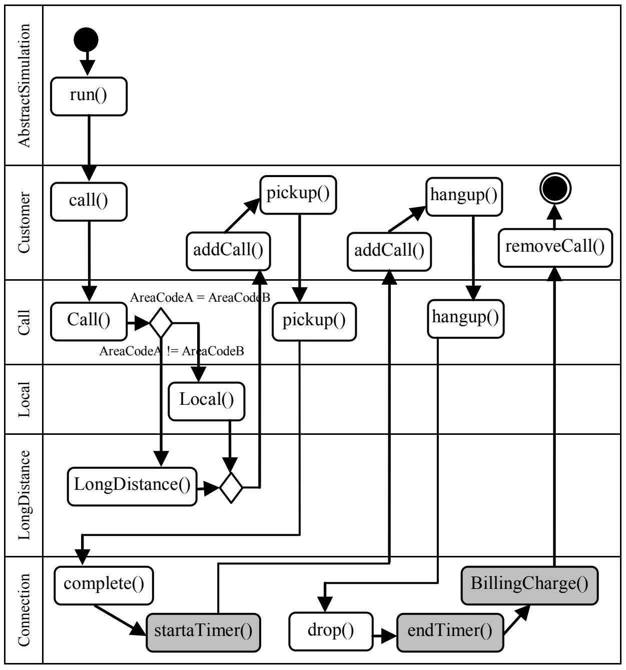
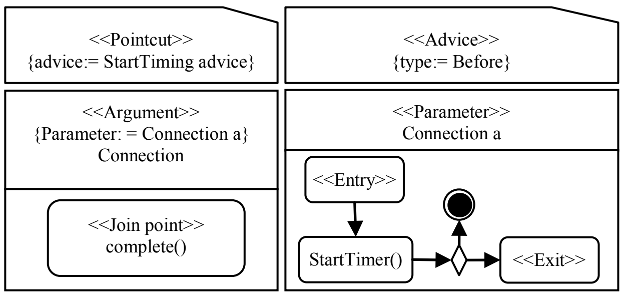
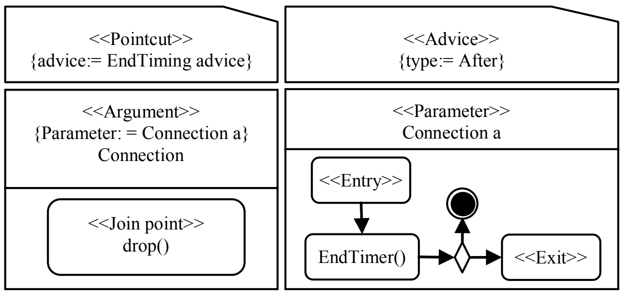


Figure 4: The integrated model with two aspects Timing and Billing

The Timing aspect in Figure 5 consists of two pointcut model StartTiming pointcut and EndTiming pointcut, and two advice model StartTiming advice for StartTiming pointcut and EndTiming advice for EndTiming pointcut. The Billing aspect in Figure 6 consists of pointcut model Billingcharge pointcut and advice model Billing charge advice for Billingcharge pointcut.

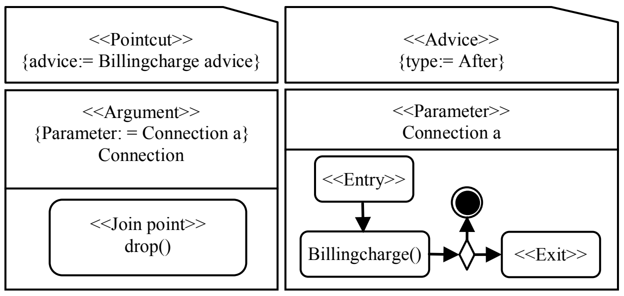


(a) StartTiming Pointcut **(b) StartTiming advice**



(c) EndTiming Pointcut **(d) EndTiming advice**

Figure 5: Timing aspect



(a) Billingcharge Pointcut **(b) Billingcharge advice**

Figure 6: Billing aspect

We created several versions of faulty AspectJ code that each version indicated one specific aspect fault. The faults included incorrect advice type, weak or strong pointcut strength, and incorrect aspect precedence. Our experiment results show that our approach is capable revealing these types of faults. Table 4 shows an example for each of the target fault types. Each row is for a specific fault type. It includes:

- The specification of advice type, pointcut, and aspect precedence.
- The expected method sequences.
- The actual implementation of advice type, pointcut, and aspect precedence.
- The actual method sequences.

Let us take the first row as an example. The expected sequence is different from the actual sequence. The difference in the sequences helps us reveal the difference between the specification and implementation and discover the corresponding fault: the advice type is changed from After (in the specification) to Before (in the implementation). The second row describes a weak pointcut fault, where Call (void Connection.complete()) is replaced with Call (void Connection.*). The specification only picks out calls to the complete method of Connection; the implementation, however, picks out calls to any method of Connection class. The third row describes a strong pointcut fault where Call (void Connection.complete()) is replaced with Call (void Connection.complet()). For both cases, we observe that exist a difference between the expected and the actual sequences and use such information to discover the corresponding faults. The fault described in the fourth row belongs to the type of incorrect aspect precedence between two aspects Timing and Billing. The implementation uses an incorrect precedence. Once again, we can discover the corresponding fault by examining the difference between the expected and the actual sequences.

Table 4. Examples for revealing different types of faults

Type of fault	**Model**		**Implementation**	
	Advice type/ Pointcut Pattern/ Aspect Precedence	Expected sequence	Advice type/ Pointcut Pattern/ Aspect precedence	Actual sequence
Incorrect advice type	After/ Call (void Connection. complete()) / NA	AbstractSimulation.run()→ Customer.call()→Call.Call()→ [Areacode A = Areacode B] Local.Local()→ Customer.addcall()→ Customer.pickup()→ Call.pickup()→Connection complete()→ Timing.StartTimer()→ Customer.hangup()→Call.hangup()→ Connection.drop()→ Timing.EndTimer()→ Customer.removecall()	Before/ Call (void Connection. complete()) / NA	AbstractSimulation.run()→ Customer.call()→Call.Call()→ [Areacode A = Areacode B] Local.Local()→ Customer.addcall()→ Customer.pickup()→ Call.pickup()→Timing.StartTimer()→ Connection complete()→ Customer.hangup()→ Call.hangup()→Connection.drop()→ Timing.EndTimer()→ Customer.removecall()
Weak pointcut	After/ Call (void Connection. complete())/ NA	AbstractSimulation.run()→ Customer.call()→Call.Call()→ [Areacode A = Areacode B] Local.Local()→ Customer.addcall()→ Customer.pickup()→Call.pickup()→ Connection complete()→ Timing.StartTimer()→ Customer.hangup()→ Call.hangup()→Connection.drop()→ Timing.EndTimer()→ Customer.removecall()	After/ Call (void Connection. *)/NA	AbstractSimulation.run()→ Customer.call()→Call.Call()→ [Areacode A = Areacode B] Local.Local()→ Customer.addcall()→ Customer.pickup()→ Call.pickup()→Connection complete()→ Timing.StartTimer()→ Customer.hangup()→Call.hangup()→ Connection.drop()→ Timing.StartTimer()→ Timing.EndTimer()→ Customer.removecall()
Strong pointcut	After/ Call (void Connection. complete())/ NA	AbstractSimulation.run()→ Customer.call()→Call.Call()→ [Areacode A = Areacode B] Local.Local()→ Customer.addcall()→ Customer.pickup()→Call.pickup()→ Connection complete()→ Timing.StartTimer()→ Customer.hangup()→ Call.hangup()→Connection.drop()→ Timing.EndTimer()→ Customer.removecall()	After/ Call (void Connection. complet ())/ NA	AbstractSimulation.run()→ Customer.call()→Call.Call()→ [Areacode A = Areacode B] Local.Local()→ Customer.addcall()→ Customer.pickup()→Call.pickup()→ Connection complete()→ Customer.hangup()→Call.hangup()→ Connection.drop()→ Timing.EndTimer()→ Customer.removecall()
Incorrect precedence	NA/ NA/ Timing, Billing	AbstractSimulation.run()→ Customer.call()→Call.Call()→ [Areacode A = Areacode B] Local.Local()→ Customer.addcall()→ Customer.pickup()→Call.pickup()→ Connection complete()→ Timing.StartTimer()→ Customer.hangup()→Call.hangup()→ Connection.drop()→ Timing.EndTimer()→ Billing.Billingcharge()→ Customer.removecall()	NA/ NA/ Billing, Timing	AbstractSimulation.run()→ Customer.call()→Call.Call()→ [Areacode A = Areacode B] Local.Local()→ Customer.addcall()→ Customer.pickup()→Call.pickup()→ Connection complete()→ Timing.StartTimer()→ Customer.hangup()→Call.hangup()→ Connection.drop()→ Billing.Billingcharge()→ Timing.EndTimer()→ Customer.removecall()

6. CONCLUSION AND FUTURE WORK

We present, in this paper, an activity-based testing approach for aspect-oriented programs. Our approach can help testers reveal several types of faults that specific to aspectual structures, such as incorrect advice type, strong or weak pointcut expressions, and incorrect aspect precedence.

Our strategy is divided into three main phases: (1) Building activity model of the primary concern and generating basic test sequences based on it. This step verifies if the primary concern is working correctly and errors, that are not aspect-related, are eliminated. (2) Building aspect models and integrating them into the primary model, in an iterative way, and generating the test sequences corresponding to them based on the defined coverage criteria. By integrating aspects incrementally, we reduce the complexity of the test and in case of failure we can precisely target the origin of the errors. (3) Verifying the generated sequences. This phase is supported by an instrumentation of the AspectJ code of the program under test. This makes it possible to check if the implementation conforms to the specification.

Currently, our approach relies on manual derivation of testing sequences from the activity models. Nevertheless, it is of interest to investigate how our approach can enhanced with automation of test sequence generation.

7. REFERENCES

[1] Filman, R. E., Elrad, T., Clarke, S., and Aksit, M., "Aspect-Oriented Software Development", Addison-Wesley Professional, Boston, 2004.

[2] Hursch, W. L., and Lopes, C. V., "Separation of Concerns", Technical Report No. NUCCS-95-03, College of Computer Science, Northeastern University, Boston, 1995.

[3] Colyer, A., Clement, A., Harly, G., and Webster, M., "Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools", Addison- Wesley Professional, 2004.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] OMG, UML Superstructure v2.1, http://www.omg.org/documents/formal/uml.htm.

[10] 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, 2004.

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

[12] Zhao, J. and Rinard, M., "System dependence graph construction for aspect-oriented programs", MIT-LCSTR-891, Laboratory for Computer Science, MIT, 2003.

[13] Zhou, Y., Richardson, D., and Ziv, H., "Towards a practical approach to test aspect-oriented software", In Proc. of the 2004 Workshop on Testing Component-based Systems (TECOS 2004), 2004.

[14] Xie, T. and Zhao, J., "A framework and tool supports for generating test inputs of AspectJ programs", In Proc. of the 5th International Conference on Aspect-Oriented Software Development (AOSD' 06), pp. 190-201, 2006.

[15] 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.

[16] Xu, D., Xu, W., and Nygard, K., "A state-based approach to testing aspect-oriented programs", In Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering, pp. 366-371, 2005.

[17] Xu, D., and Xu, W., "State-based incremental testing of aspect-oriented programs", In Proceedings of the 5th International Conference on Aspect-Oriented Software Development, pp. 180-189, 2006.

[18] Xu, W., and Xu, D., "State-based testing of integration aspects", In Proceedings of the 2nd Workshop on Testing Aspect-Oriented Programs, pp. 7-14, 2006.

[19] Binder, R. V., "Testing Object-Oriented Systems: Models, Patterns, and Tools", Addison-Wesley Professional, Boston, 2000.

[20] Xu, W., Xu, D., and Wong, W. E., "Testing Aspect-Oriented Programs with UML Design Models", International Journal of Software Engineering and Knowledge Engineering, Vol. 18, No. 3, pp. 413-437, May 2008.

[21] 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, 2005.

[22] Liu, C. H., and Chang, C. W., "A State-Based Testing Approach for Aspect-oriented Programming", In Journal of Information Science and Engineering , pp. 11-31, 2008.

[23] Badri, B., Badri, L., Fortin, M. B., "Automated State-Based Unit Testing for Aspect-Oriented Programs: A Supporting Framework", In Journal of Object Technology, vol. 8, no. 3, pp. 121-126, 2009.

[24] Cui, Z., Wang, L., and Li, X., "Modeling and integrating aspects with uml activity diagrams", Proceedings of the 2009 ACM symposium on Applied Computing, 2009.

[25] Mortensen, M., and Alexander, R., "Adequate Testing of Aspect-Oriented Programs", Technical report CS 04-110, Colorado State University, Fort Collins, Colorado, USA, December 2004.

[26] Offut, J., Xiong, Y., and Liu, S., "Criteria for Generating Specification-based Tests", In Engineering of Complex Computer Systems, ICECCS '99, 1999.

[27] Offut, J., and Voas, J., "Subsumption of Condition Coverage Techniques by Mutation Testing", ISSE-TR-96-01, January 1996.

[28] Beizer, B., "Software Testing Techniques", International Thomson Computer Press, 1990.

[29] AspectJ Web Site, http://eclipse.org/aspectj/.