ABSTRACT

MODELING AND TESTING OF ASPECT-ORIENTED SYSTEMS

by Wenhuan Jiang

April, 2011

Director of Thesis: Junhua Ding

Department of Computer Science

Aspect-Oriented programming modularizes crosscutting concerns into Aspects, which are automatically weaved to the specified points of a program. Although Aspect-Oriented programming improves program maintainability and the encapsulation of crosscutting concerns, it also breaks some traditional programming rules due to the weaving mechanism. Therefore, a new software testing approach has to be developed to rigorously test Aspect-Oriented programs. In this thesis, we introduce the concept of Aspect modeling and Aspect testing and then proceed to our investigation of a model-based incremental approach for testing Aspect-Oriented programs. First, a state machine model in UML is created for each Aspect and each base class, which is the class to be weaved with the aspect. Then each individual aspect or base class is tested using the test cases generated from state machine models. A combined state machine model is established by weaving the aspect model into the base class model. Finally, we perform a test on the woven program using test cases generated from the combined state machine model. Because the number of scenarios for weaving aspects and base classes could be very large, it may require a huge number of test cases to effectively test the program. To speed up the process, we propose a prioritizing strategy for selecting test cases in order to find errors sooner since different test cases have different capacity for tracking errors. We demonstrate that the test cases generated from the state machine model have to satisfy the adequacy of the transition coverage, the round-trip coverage, and the state coverage in the state machine model. Furthermore, the prioritizing strategy is developed based on the number of changes brought by weaving of an aspect and its base classes. The test case including more changes will have a higher priority. The effectiveness of the investigated strategy is evaluated through the case study and the mutation testing. The result of case study shows that the model-based incremental approach integrated with prioritizing test case selection provides an effective tool for testing large-scale Aspect-Oriented systems.

Modeling and Testing of Aspect-Oriented Systems

A THESIS

Presented To

The Faculty of the Department of Computer Science

East Carolina University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Wenhuan Jiang

April, 2011

MODELING AND TESTING OF ASPECT-ORIENTED SYSTEMS

by

Wenhuan Jiang

APPROVED BY:

DIRECTOR OF THESIS: ______________________________

Junhua Ding, PHD

COMMITTEE MEMBER: ______________________________

M. N. H. Tabrizi, PHD

COMMITTEE MEMBER: ______________________________

Masao Kishore, PHD

COMMITTEE MEMBER: ______________________________

Sergiy Vilkomir, PHD

CHAIR OF THE DEPARTMENT OF COMPUTER SCIENCE:

____

Karl Abrahamson, PHD

DEAN OF THE GRADUATE SCHOOL:

____

Paul J. Gemperline, PhD

Acknowledgements

First of all, I want to take this opportunity to thank the computer science department for their help and support.

I am indebted to my advisor, Dr. Junhua Ding, for the guidance and patience he has provided since I started working under his supervision, and for his constant encouragement without which I would not have pursued a master’s degree in computer science.

I am also grateful to Dr. Tabrizi, director of graduate students in computer science department, for his helpful advice. Many thanks to Dr. Kishore, Dr. Vilkomir and Dr. Tabrizi for serving on my thesis committee, reviewing this draft, and for making sure I met all the graduation requirements set by the university. My classmates and friends in the computer science department have encouraged me to go through this hard time and in this respect, I would like to express my appreciation to Tong Wu.

Finally, I owe my gratitude to my parents for their continuing support to further education and my wife Theresa Hua, for her company throughout these years.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

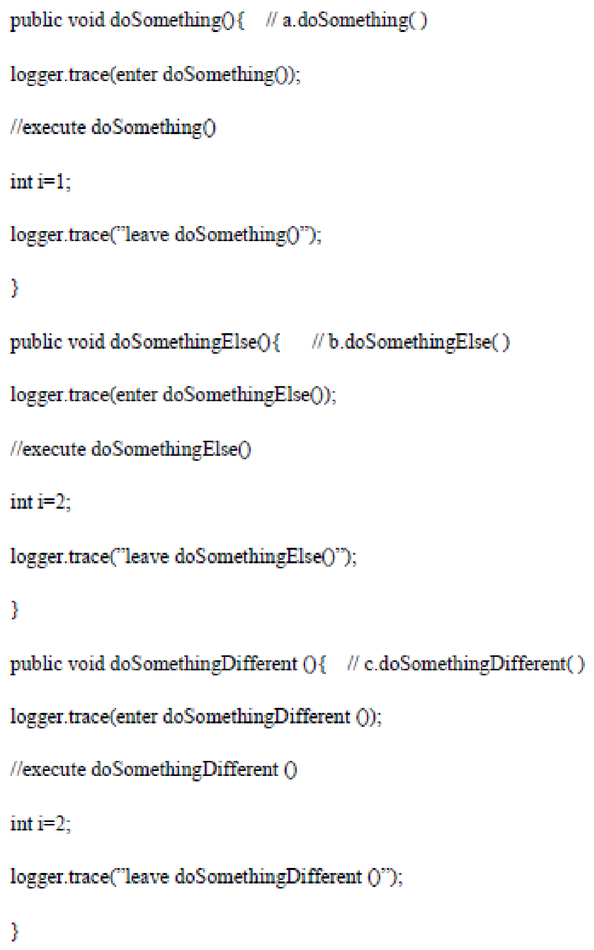
CHAPTER 1 INTRODUCTION

Aspect-Oriented Programming (AOP) has attracted research interests in software engineering in recent years for its potentials in development of large software systems and is the main topic of this thesis research. In this chapter we will introduce the background and basic concepts related to AOP, which include the concept of AOP, state-based modeling and software testing.

1.1 AOP and related concepts

In software engineering, a concern is a problem that the software is designed to solve. For example, database interaction, numerical calculation, conversation, etc are targeted problems for the design of certain software, which lead to the commercial development of SQL Server, MatLab, and MSN. A typical software system consists of core concerns that have one primary or system-wide behavior and other concerns that cut across the primary decomposition of the system [1]. Although those crosscutting concerns are not related to core concerns directly, they are required for proper execution of the program. For example, a bank system has a core concern of processing the transaction such as depositing, withdrawing, and managing the account. Crosscutting concerns like implementing logging are commonly distributed across classes and methods. Although logging is not directly related to processing transactions, it requires the execution after successful processing according to the software design. Such crosscutting concerns affect several implementation modules, such as account modules, system modules, etc. Using the conventional method of Object-Oriented Programming (OOP) to implement crosscutting concerns may cause code scattering and tangling, which means the code related to the crosscutting concerns is implemented throughout the entire source code or woven within the code implementing the local module.

To illustrate the above problem, let's consider an example of code scattering and tangling. There are three classes *a*, *b*, and *c*, and three related functions '*a.doSomething*( )', '*b.doSomethingElse*( )', and '*c.doSomethingDifferent*( ) '. In the following approach based on the OOP (see Figure 1), logging instructions ('*logger.trace*( )') are in charge to save the current operations in the log, which is the record of all instructions listed by time or other factors. Although Logging functions are not core concerns ('*doSomething*', '*doSomethingElse*', and '*doSomethingDifferent*' are core concerns), they are still important to the completeness of the system functions. Thus, they have to be implemented properly in the program. But using conventional OOP method can be problematic. As shown in the program code of Figure 1, logging instructions are scattered throughout the methods of different modules as shown by two '*loger.trace*' functions in each class. Thus in each method, logging instructions and functional code are tangled and '*loger.trace*' functions are next to variable assignment instruction '*int i=1*'. The scattering and tangling as shown in this simple example is awkward and in many cases problematic for software modeling, testing, and maintaining. The above problem can become extremely severe for development of modern large software programs which are much larger than the illustrating example. For example, what will happen if we have more than fifty classes implementing logging instructions in a large system? The consideration has led to the development and use of an improved software design to reduce the extent of tangling and scattering since they can't be completely removed. The cost of the design improvement is also a factor in its implementation.



```
public void doSomething(){    // a.doSomething( )
logger.trace(enter doSomething());
//execute doSomething()
int i=1;
logger.trace("leave doSomething()");
}
public void doSomethingElse(){     // b.doSomethingElse( )
logger.trace(enter doSomethingElse());
//execute doSomethingElse()
int i=2;
logger.trace("leave doSomethingElse()");
}
public void doSomethingDifferent (){    // c.doSomethingDifferent( )
logger.trace(enter doSomethingDifferent ());
//execute doSomethingDifferent ()
int i=2;
logger.trace("leave doSomethingDifferent ()");
}
```

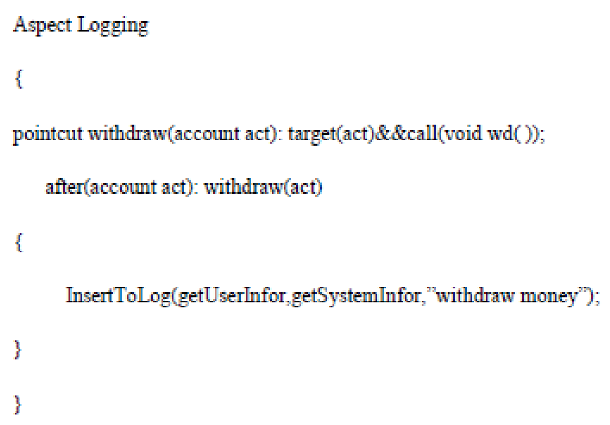
Figure 1: Example of tangling and scattering

AOP is a software design method developed for reducing code tangling and scattering in a program by separation of concerns in software development. In AOP, a crosscutting concern is modeled and implemented separately as a single module called Aspect, which is a new type of module different from regular classes.

AOP implements crosscutting concerns in Aspects with a special mechanism. In each Aspect, there are advices invoked at certain points of program execution. Here is an analogy to our daily life to understand the meaning of the Aspect concept. Execution of a software program can be treated as driving a vehicle on a road. Aspect can be used as a GPS device for guidance of the trip. When the vehicle (program execution) reaches the certain point, the GPS provides the advice about next direction, which means Aspect provides guidance to the execution of a dynamic task instead of the static one. New constructional elements (different from those in the conventional OOP) are introduced in Aspects. In the AOP approach, Aspect is developed as a container to possess elements such as *pointcuts* and *advices*. Their relation can be interpreted as a class holding methods and attributes. An AOP based project can have several classes and Aspects. Within an Aspect, a *join point* is a location in the code which is defined by users and where a concern can crosscut. It specifies the location for execution of different *advices*. *Join point* can be any of the following: method calls, method executions, constructor calls, and constructor executions. One or several *join points* can be combined to form one *pointcut*.

An *advice* is the part of a code which is executed when a *pointcut* is matched. An *advice* can be executed before, after, and around a particular *join point*. It is quite similar to an event-driven database trigger.

The most well known AOP system is AspectJ developed as an Aspect-Oriented extension for Java [2]. In AspectJ, one can use Java language to define regular classes, and use a Java-extension language to define Aspects. To better handle the crosscutting concerns, AspectJ provides new constructions. Base classes are abstract units of core concerns and Aspects are abstract units of crosscutting concerns. The code shown in Figure 2 provides an example of the AspectJ program. Aspect '*Logging*' adds the log information to the system after withdrawing money. Class '*account*' has an instance '*act*' and' *wd( )*' is a function of '*account*', which is also defined as a *join point*. *Pointcut* '*withdraw*' is defined by the set of *join points* to handle when and where to add a log. The rest of the program starting with a key word '*after*' can be treated as the *advice*, where the function '*InsertToLog*( )' is called to add a log after '*wd*( )' is executed.



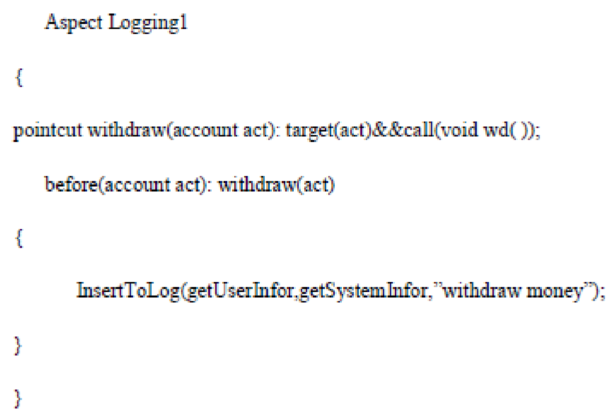
```
Aspect Logging
{
pointcut withdraw(account act): target(act)&&call(void wd( ));
    after(account act): withdraw(act)
{
      InsertToLog(getUserInfor,getSystemInfor,"withdraw money");
}
}
```

Figure 2: Aspect Logging

As discussed and demonstrated above, AOP changes the conventional development process of software by developing and testing classes and some basic concerns at the beginning of the software. After that Aspects are designed and woven into the previous classes with both core and crosscutting concerns. AOP improves separation of concerns and makes it possible to create models for crosscutting concerns.

1.2 Testing of AOP

Despite the benefits as discussed previously, implementation of AOP in software development generates new challenges for modeling and testing. With the use of *join points* and *advices*, Aspects show where and how a concern is addressed. In a static program, an Aspect is located at certain position of the code. When the *advice* is dynamically executed, however, it can be executed at any *join point* in classes. Figure 3 presents an example of the Aspect fault. In Aspect Logging1, the *advice* is different from the one in Aspect Logging shown in Figure 2 since '*before*' is defined instead of '*after*.' Although the change is only one word, the consequence is altered after weaving. When the program is about to run the '*withdraw*' function, it executes the *advice* and insert the '*withdraw*' operation into log. Then it runs the '*withdraw*' function after saving the log. This procedure is invalid if the '*withdraw*' fails because the log of the operation is saved at first but the operation does not execute correctly. *Advices* are not the only source of faults and more faults can be found in *pointcuts* definitions which have more connections with classes. With this example, we can conclude that inappropriate use of Aspects can have severe consequences to create errors in the software development. Furthermore, traditional testing techniques such as those for testing OOP cannot be directly used for AOP testing unless some adaption are made or certain extension mechanisms are added for testing AOP applications.



```
    Aspect Logging1
{
pointcut withdraw(account act): target(act)&&call(void wd( ));
    before(account act): withdraw(act)
{
        InsertToLog(getUserInfor,getSystemInfor,"withdraw money");
}
}
```

Figure 3: A fault of Aspect Logging

1.3 State-based Testing

Different methods can be used to test Aspect faults. In this thesis research, we investigate an approach termed as State-Based Testing (SBT). SBT is a software testing method based on the Finite-State Machine. In this section, we will introduce the concept of Finite-State Machine and discuss the implementation of Aspects in the Finite-State Machine followed by our strategy to prioritize SBT.

1.3.1 Finite-State Machine (FSM)

FSM is a behavior model used to analyze and design the software composed of a finite numbers of states, events, and transitions. Once the model is implemented, the operation of the FSM starts from the initial state and goes to different states via transitions. We chose an extended state machine in UML to model AOP programs.

Figure 4 shows the state model of a connection class. We define different states according to its state variables (return values of getState( )) such as

State P (Pending): getState( )=0

State C (Complete): getState( )=1

State D (Dropped): getState( )=2

Events in connection class include connection, complete, drop, connects, and getting States. Three states capture all relationships based on variable in each state, as shown in Figure 4.

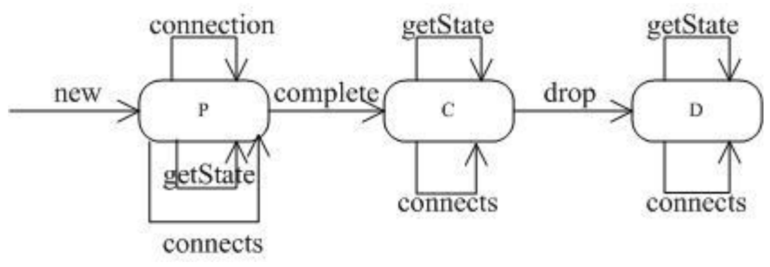


Figure 4: State model of connection class

1.3.2 Different types of Advices *implemented in FSM*

FSM is not only used in Object-Oriented Modeling (OOM) but also used to implement AOM. Different types of Aspects can be implemented accurately in state models. We use the state model to fit the Aspect weaving mechanism.

Advice types in the state model decide the connection of an advice and related join points. Four different types of advices are before, after, around, and concurrent. We illustrate four types of advices with the following example in which the base class changes states from *S1* to *S2* by calling the method *m1*. Here *m1* is defined as the join point. Figure 5 shows a Base Class State Model.

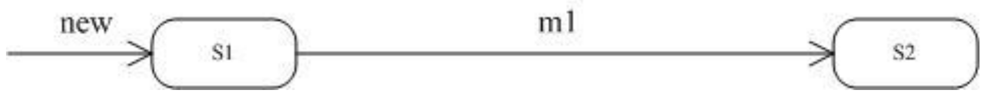


Figure 5: FSM for Base class

In the Aspect, *ma* is the method in advice and *SA* is the new state after weaving the Aspect.

1) Advice type before.

As shown in Figure 6, *ma* was called before *m1* by the advice in its Aspect. Then the object changes the state into *SA*. Depending on the attributes it modifies, *SA* may be the same or different with *S2*.

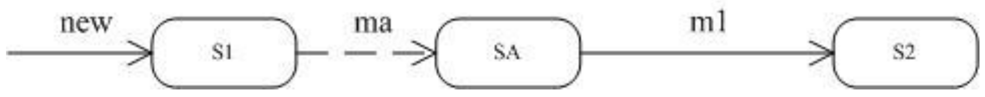


Figure 6: FSM for Advice type before

2) Advice type after.

Figure 7 presents a case in which the object changes into state *S2* after calling *m1*. Then it calls *ma* in the advice of the Aspect and finally arrives at state *SA*.

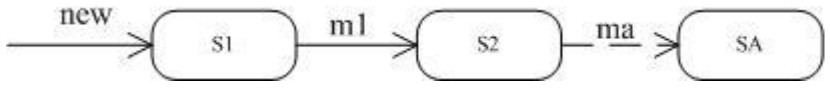


Figure 7: FSM for Advice type after

3) Advice type around.

When the object intends to call method *m1*, method *ma* in advice is called instead. Then the object changes from state *S1*to state *SA* without calling method *m1* as indicated by the red cross. Consequently, state *S2* is not reached, as shown below.

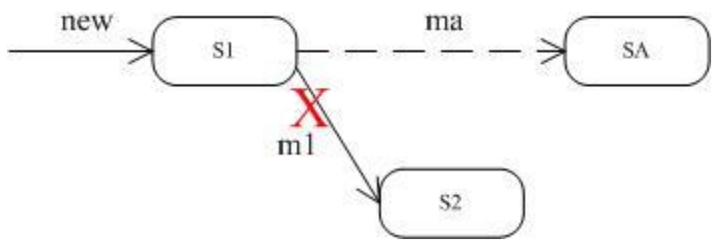


Figure 8: FSM for Advice type around

4) Advice type concurrent.

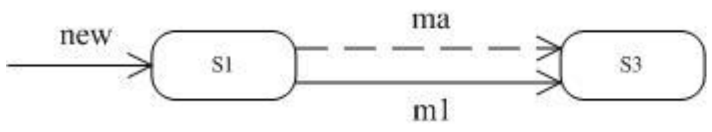


Figure 9: FSM for Advice type concurrent

Figure 9 illustrate a case that when the object calls method *m1*, method *ma* in advice is called simultaneously. A new state *S3* is reached from state *S1*.

Many advices can be woven with the same join point, which is referred to as a Shared Join Point (SJP). The composition of different Aspects at the SJP generates many issues such as the dependency between Aspects and the execution orders. There are four different relations between Aspects at the SJP. One can use the state model in Figure 5 as the base state model the advice type before to show four different relations. In this case *MA* is the method in advice of Aspect *A* and *MB* is the method in advice of Aspect *B*. *SA* is the state after weaving Aspect *A* and

SB is the state after weaving Aspect *B*. Here *A* and *B* are two different Aspects which can be composed with the same join point.

A||B means the sequence of execution between *A* and *B* does not matter. Aspect *A* and Aspect *B* can be executed at the same time or not at the same time. The sequence of execution *A* and *B* does not affect the result of Aspect state models. A->B means *B* can never be executed until *A* has been executed. Figure 10 shows that before join point *m1* method *MA* is called at first, and then *MB* is called from state *SA*.

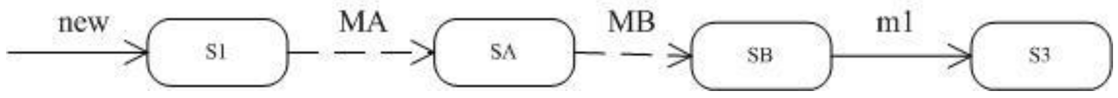


Figure 10: FSM for A->B

A-B means whether *B* will be executed or not and this is decided by the results of *A*. Aspect *A* executes at first. In state *SA*, the object checks the attribute in *SA* and decides whether it will call *MB* or call *m1* directly.

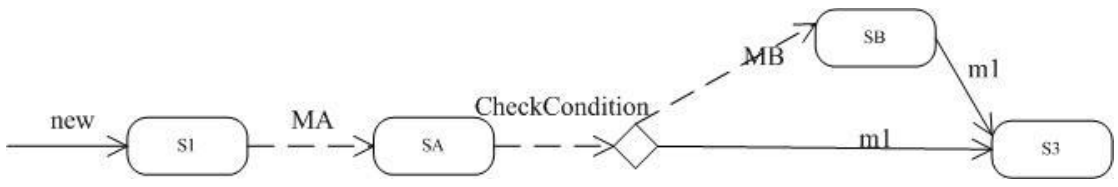


Figure 11: FSM for A-B

A|B means *A* or *B* will execute and Figure 12 shows an example of A|B.

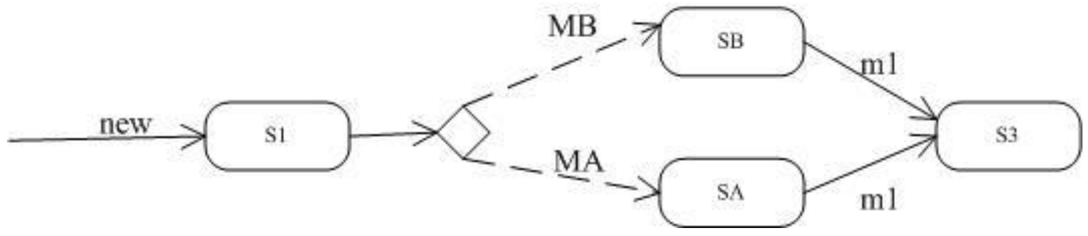


Figure 12: FSM for A|B

From the above definitions of relations, it is obvious that the relation is transitive. For example, if we have A->B and B->C, then we will have A->C.

The execution order of the Aspect can be solved by the above definitions, but the dependency between Aspects is not obvious. The solution is to check each SJP and build a dependency path based on execution sequences.

In Chapter 3, we will provide more details about how our state model and woven model are defined.

1.3.3 Prioritizing SBT

FSM is rigorous in the design phase, because it covers all objects, interactions, and possible activities in the design. SBT is a framework for rigorously testing design models. With SBT, an executable model is tested with sequences as test cases, running before the actual implementation. SBT also automates the test generation from state models [3]. Several studies have demonstrated that model-based testing improves the fault detection capability and reduces testing costs [4]. State-based approach generates test sequences from all states of each object in the system, and guarantees the testing coverage. All above advantages make SBT a reliable approach. AOP can be tested in an incremental way [5]. For a given AOP, we can first test the base classes. If there is a fault in the base class, it can be found alone at the very beginning. Then we test the base classes and Aspects as a single unit. The benefit of the whole process is that it can help testers find Aspect faults.

However, the state-based approach may suffer from the state explosion problem. A software system may have many classes and Aspects, and the number of weaving scenarios of these classes and aspects could be very large. Therefore, adequately testing those scenarios may need too many test cases to be handled.

Test execution only stops when a test reports a failure. The time of waiting is longer than we expected. The solution to solve this problem is to prioritize the Aspect testing, and then the test will report a failure earlier, reducing the test execution time. In this thesis research, we present an approach to test AOP before prioritizing and after prioritizing. We prioritize Aspect testing by identifying new elements generated from Aspects.

The rest of this thesis is organized as follows. We will review additional research literature related to the AOP in Chapter 2 and introduce the concepts of AOM and state-based modeling in Chapter 3. Our modeling includes the state model for core concerns and the state model for crosscutting concerns. In Chapter 4, we present the testing and related prioritization process. Chapter 5 presents two case studies with conclusions presented in Chapter 6.

CHAPTER 2: RELATED WORK

Investigations on AOP testing can be separated into two groups according to their execution at either the implementation level or the model-based level. At the implementation level investigations have been focused on testing of unit, integration, system and related automatic tools. At the model-based level fault models and design-based model testing have been developed such as flow-based and state-based testing. The different methods can be combined to study single parts inside an Aspect such as testing pointcut and pointcut descriptions. This chapter provides a detailed overview of these testing methods.

2.1 AOM

France et al. [28] present an approach that simplifies the task to address concerns. The study produces Aspect-Oriented architecture models (AAMs) that describe how concerns are addressed. An AAM consists of a set of Aspect models and a base architecture model. The composition of the Aspect and base model in an AAM produces an integrated view of the logical architecture described by AAM.

An Aspect mapping method has been developed by Sanchez et al. [29] to incorporate code requirements into architectural design, which includes algorithms to guide appropriate arrangement of Aspect properties during software development. This approach allows Aspect decisions captured at each stage of development to be modified in a later stage. It also provides a means to record decisions based on the alternative decisions and possible modification.

Klein et al. [30] propose some interpretations for pointcuts that allow multiple behavioral Aspects to be statically woven. This woven allows join points to match a pointcut even when some extra-messages occur. They also introduce a formal definition of new merging operator and describe its implementation. Mussbacher et al. [31] demonstrate how scenario-based Aspects can be modeled at the requirements stage with the help of Use Case Maps (UCMs). UCMs are a visual scenario that model Aspects and pointcut expressions in a visual way. It is difficult to analyze requirements and locate all points in the system. To solve this problem and identify Aspect, Baniassand et al. [32] suggest Theme approach for viewing the relationships between behaviors in requirements and identifying Aspects in requirements.

2.2 Unit testing and Integration testing

Zhao et al. [7] advise a data-flow-based unit testing approach for AOP. Their approach individually tests Aspects and classes and performs three levels of testing for each Aspect or class including intra-module, inter-module, intra-Aspect and intra-class testing. Control flow graphs are used to compute *def-use* pairs of an Aspect or class. And the *def-use* pairs are used to decide the selection of tests for Aspects or classes.

Another method is presented by Zhou et al. [10] to use a comprehensive methodology that can effectively test Aspects. It adapts traditional unit testing, integration testing, and system testing to test AOP. To reduce the cost of testing Aspects, test cases developed for regular classes can be reused. An algorithm is also defined based on control flow analysis to select relevant test cases that test the Aspects. A tool support for their method is also developed.

Lemos et al. [9] develop a derivation of a control and data flow model for Aspect-Oriented programs, which are based upon the static analysis of the object code resulted from the weaving process. Its first step is applying the structural testing technique to AOP. Their approach supports the unit testing, pieces of advice in isolation, and the interaction among pieces of advice.

A framework called Aspectra is introduced to automatically generate test inputs for testing AspectJ program [10]. It leverages existing test-generation tools to generate test inputs for the woven class. To enable Aspects to be exercised during test generation, Aspectra automatically synthesizes appropriate wrapper classes for woven classes and measures Aspectual branch coverage to assess the quality of the generated tests. Tool support has been developed for automating Aspectra's wrapper synthesis and the coverage measurement.

An approach is proposed by Xu et al. [11] to automatically generate the unit testing framework and test oracles from Aspects in AOP. They introduce a concept called application-specific Aspects. An Aspect-Oriented Test Description Language and techniques are described to build top-level Aspects for testing generic Aspects. Runtime exceptions thrown by testing Aspects are used to decide whether methods work well. Finally they use a double-phase testing way to filter out meaningless test cases in the framework.

A language-centric approach is also proposed to automatically test the adequacy and analysis the coverage of concerns [12]. This approach allows the tester's intentions to be represented abstractly within source code. A white-box join point model and a generalized action framework to support testing tools are provided.

Deursen et al. [13] suggest a refactoring and testing strategy. It supports a systematic approach and considers issues of the behavior conservation and integration of the Aspect. Their strategy is applied to an open source project called JHotDraw and illustrated on a group of selected concerns of the model and features of the employed Aspect.

Based on various types of control flow graphs that can be used to select from the original test cases that execute changed code for the new version of the AspectJ program, Zhao et al. [14]

present a regression test for AOP. Their technique can be used to modify the individual Aspect or class as well as the whole program that uses modified Aspects or classes.

2.3 Model-based test

Comparing with Unit testing and Integration testing, Model-based testing provides an appealing approach because of the following benefits [5] ：1. The modeling activity helps clarify requirements and build a connection between developers and testers. 2. Some design models can be reused for testing purposes. 3. Model-based testing process can be or partially be automated. 4. Model-based testing can reduce testing cost and improve the detection ability by generating and executing many test cases automatically. Fault models and Design-based models are two major Model-based tests.

2.3.1 Fault models

In model-based testing, fault models and mutant testing are crucial. Sometimes they are used to determine the testing ability of other model-based approaches. Alexander and Bieman [15] introduce a candidate fault model with associated testing criteria. The candidate fault model consists of six faults, which include incorrect strength in pointcut patterns, incorrect Aspect precedence, failure to establish expected postconditions, failure to preserve state invariants, incorrect focus of the control flow, and incorrect changes in control dependencies. Although their work does not constitute a fully developed testing approach, the fault model they present helps others develop the effect approach to the systematic testing of AOP.

Maldonado et al. [6] propose a set of mutation operators for AOP. Since mutation operators are essential for the evaluation of testing approaches, they design a set of mutation operators for the AspectJ program. The operator models which are instances of fault types are created according to existing works of fault types. Aspect fault types include pointcut faults, inter-type faults, advice faults, and base program faults. They also mention that most fault types occur during the implementation phrase.

Prasanth et al. [16] introduce a framework that automatically generates mutants for a pointcut and identifies mutants that are similar with the original expression. Their framework generates only relevant mutants rather than arbitrary strings, which helps developers save manual efforts in identifying equivalent mutants and generating efficient mutants.

Mutant test is also used in defining the adequate testing. Mortensen et al.[17] presents a framework for defining the adequate testing of AOP. They combine the white box coverage and the mutation testing to adequately test AOP. A static analysis of an Aspect is used to guide in the selection of white box criteria. Meanwhile, they provide a set of mutation operators to evaluate if a test suite is adequately testing the new code fragments. In our work, we also use the mutation test to find out if our approach is efficient to test Aspect faults.

2.3.2 Design-based model

Some researchers focus on designing the Aspect-Oriented model with UML mechanism, which can be used as a guide for future testing. Stein et al. [18] describe a design model for the development of AspectJ programs with the UML. It extends UML concepts and uses standard UML extension mechanisms to provide Aspect-Oriented concepts as they are defined in AspectJ. The weaving mechanism can be represented with existing UML concepts.

Zhang et al. [19] present High-Level Aspect (HiLA) for UML state machines to address the synchronization or execution history dependence. HiLA specifies multiple history-dependent and concurrent Aspects that extend the behavior of a base state machine in a straightforward way.

An approach to define the symmetric AO model and enhance the model by testing individual concerns through simple tests is offered by Jackson et al. [20]. They also define a merge operator that synchronizes the merging of both models and tests.

Inspired from the Aspect-Oriented Modeling, Xu and his team have done a great deal of work in model-based testing. One of their work is [21], an approach to generating tests for adequately exercising interactions between Aspects and classes based on UML models. Their model is composed with class diagrams, Aspect diagrams, and sequence diagrams. The advice on a given method can be woven into a new sequence diagram. The approach constructs a flow graph from the woven sequence diagram for a given coverage criteria and expends the graph to a flow tree. Each path of the flow tree is a test.

Xu and his team introduce an approach of AOM and verification with finite state machines [22]. In their work, they provide explicit notations such as pointcuts, advices, and Aspects with state models. They compose Aspect models and class models in an AOM through a weaving mechanism. Then they transform the AOM and class model, which is not affected by the Aspects into Finite State Processes. These are to be checked by the model checker based on the desired system properties.

An approach of testing integration Aspects is also illustrated by Xu et al. [23]. They use Aspect-Oriented state models for specifying integration Aspects and compose state models of Aspects and classes. Then they generate test cases for integration Aspects from their state models. The integration Aspects are exercised through the interface of their base classes. In their following work, they treat Aspect testing as an incremental process in the state-based approach [5]. Aspects are considered as incremental modifications to their base classes. An AO extension to state models is built, which helps specify the impact of Aspects on the states and transitions of base class and the generation of test cases. Meanwhile, their work shows that the majority of base class tests can be reused for Aspects, but subtle modifications are necessary.

State-based testing can also be used to combine with other methods to generate a more reliable method. For example, Xu et al. [24] combine state models and flow graphs as an Aspect scope coverage model (ASCM) for producing test suites. They merge the class state model and the Aspect state model into ASCM that allows us to trace the behavior of AOP by identifying sequence results of the state transitions. Transitions between class and the Aspect, and corresponding actions are substituted by the advice and method flow graphs to construct an Aspect flow graph. For the generic collection of the behavior model, a transition tree is generated in terms of ASSM. Using AFG and the transition tree, the executable code-based test suites can be refined more accurately. Since a state-based approach often suffers from the state explosion problem, identifying the paths or test suites are most important. To resolve this problem, Xu et al. [25] suggest to prioritizing Aspect tests by identifying the extent to which an Aspect modifies its base classes. The modification is measured by the number of new or changed components in state transitions. Transitions with more changes have higher priorities for test generation. To evaluate the effect of Aspect test prioritization, they use mutation analysis on AspectJ programs.

2.4 Others

Some researchers are finding methods to test pointcut descriptions. Unlike testing the whole Aspect and its related classes, testing pointcut is based on the assumption that the advice and other elements in Aspect are defined accurately.

Ferrari et al. [26] present a fault classification for Pointcut Descriptors (PCDs) with a strategy to test unintended and neglected joint points adapting structural and mutation testing. They classify the types of faults that can occur in PCDs in terms of selected join points. They have a two-step strategy including helping the tester identifying extra join points selected by PCDs and identifying neglected join points that should be selected by PCDs.

Anbalagan et al. propose an automatic framework called APTE that test pointcuts in AspectJ programs [27]. APTE receives a list of source files including the source of Aspects and target classes. APTE outputs a list of matched join points and a list of boundary join points, which are events that do not satisfy a pointcut expression but are close to the matched join points.

CHAPTER 3: ASPECT-ORIENTED MODELING

Although AOP originally started at the programming level, the development of modeling is still in need. It assures the quality of an Aspect-Oriented System. The development of Aspect-Oriented System is extended over phases before the implementation of software, such as software analysis and software design. In this paper, we discuss more about the design level. Firstly, we introduce an overview of AOM, and then we introduce more details of modeling with state machines.

3.1 Overview of AOM

In general, a valid Aspect-Oriented model should explicitly express concerns and their weavings in a certain language. This certain language can be a modeling language or a programming language. In our work, we use an extension of UML. More details about using UML to model Aspect can be found in Stein's work [33].

Concern, as mentioned in chapter 1 is an interest of the system. Two types of concerns are non-crosscutting and crosscutting concerns. The non-crosscutting concern is also called base. Aspects themselves can act as a base for other Aspects. Aspect-Oriented System development generates a woven model of the system including composition of bases and Aspects or Aspects with other Aspects. There are two ways of weaving Aspects into other concerns. One is dynamic weaving, the other is static weaving. Dynamic weaving only happens when it is running. Static weaving happens at the design phase. In this paper, we work on static weaving. To guide weaving, we need a rule for Aspects to adapt at certain points of concerns which specifies where and how to adapt.

There is an example to illustrate how the above elements are presented in AOM. Theme approach supports for AOM at both requirement and design level [32]. Theme is divided into two segments Theme/Doc and Theme/UML. Theme/Doc identifies at the analysis phase. Theme/UML uses UML to model structure and behavior for a different theme at the design phase. A theme is an element of design, which represents one feature from structures or behaviors. Thus a theme points out a concern. There are two kinds of themes: Base themes and Crosscutting themes, corresponding to bases and Aspects. Using the Theme/Doc tool, we can find relationships of behaviors and decide which behaviors are bases and which are crosscutting concerns.

3.2 Aspect-Oriented Modeling with state machines

In this part, we introduce class state models and Aspect state models. At last we show the woven model.

3.2.1 Class State Models

Finite state models are often used for OOM. Recently, it has been used in Aspect modeling and testing because it is able to specify the impact of Aspects on base class objects. Now we define a state model that can weave Aspects.

A state model consists of states, events, and transitions. We define state model with three elements: S, E, and T. S is a finite set of states, E is a finite set of events, and T is a set of transitions. To explain each concept in our approach, we use programs that will be tested in our case study as an illustration. Software testing is implemented in the AspectJ project: Student

Information Management System (SIMS). SIMS is an information management system to manage student's basic information including searching and manipulation functions.

One of modules in SIMS is AddInf, which adds student information into the system then saves it to the database. In the following passage, AddInf class and its woven state model will be used for illustration.

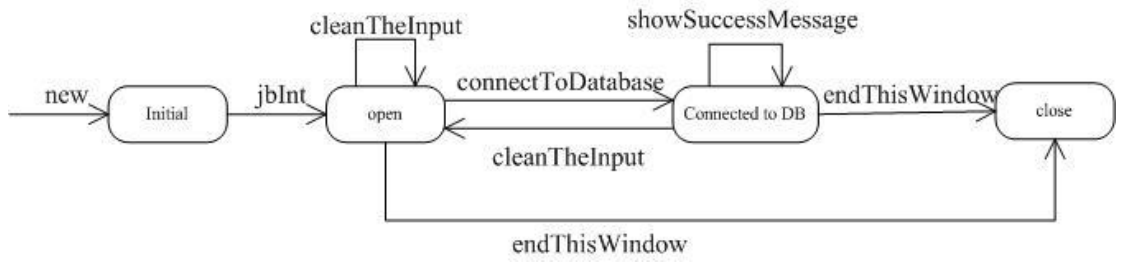


Figure 13: State model for class AddInf

There are four states for class AddInf which are on following list with their state values in Table 1.

State	Values	Functions operating on transitions
Initial	State=0	new
open	State=1	jbInt, cleanTheInput
Connected to DB(CDB)	State=2	connectToDatabas, showSuccessMessage
close	State=3	endThisWindow

Table 1: Values and functions for state in class AddInf

All transitions in AddInf state model are listed as following:

T1 (a, new, Initial)

T2 (Initial, jbInt, open)

T3 (open, cleanTheInput, open)

T4 (open, connectToDatabase, CDB)

T5 (open, endThisWindow, close)

T6 (CDB, showSuccessMessage, CDB)

T7 (CDB, cleanTheInput, open)

T8 (CDB, endThisWindow, close)

3.2.2 Aspect State Models

Base, crosscutting concern, and crosscutting relationships are fundamental elements in Aspect modeling. The base is described in state models. To specify crosscutting and relationships, we represent basic notions of AOP in Aspect State Models: joint points, pointcuts, and advices. To make it simple, other AOP notions such as Aspect inheritance and Aspect composition will not be discussed in this paper. An Aspect model includes declaration of new transition, pointcuts for state or transition, and advice models. Declaration introduces new transitions to the base model. The advice describes the control sequences to each join point in a state model.

In SIMS, the requirements for class AddInf have been changed. Three concerns are added as following:

1) When we interact with the system, we want to keep the record of these operations for security purposes. Logs should be kept after logging the system, adding new student

information, modifying the information, and deleting the student information. The log record in the database should include user ID, time, and their operations.

2) For security purposes, the operation between logging to the system and updating the student information should have a time restriction. For example, we define this restriction as 8000 milliseconds, if we have logged into the system for 9000 milliseconds and try to delete the student's information, this operation must be forbidden due to the timing issue.
3) It is normal if we change the database file or database address especially when SIMS is sold to different customers and they will use it in their own domain. Sometimes, it is necessary to operate the student information in different databases in one SIMS. Thus, changing the database without changing the original code will be helpful for developers and users.

Concerns above are difficult to be captured in original code due to their cross-cutting the system's basic functionality. And the tangled code is extremely difficult to maintain. We implement the above concerns in three Aspects. Figure 14 shows the Aspect state model for Log. Aspect DataBaseAccess and Aspect Timing are introduced in Figure 15 and Figure 16.

1) Aspect Log.

After the system shows a success message, Log implements the advice to insert the add operation into log.

Declare AddInf(CDB, InsertInToLog, CDB) // declare new transition in **AddInf**

pointcut ***addStudent*** (CDB, showSuccessMessage, CDB):AddInf(CDB, showSuccessMessage, CDB)

advice ***addStudent:***

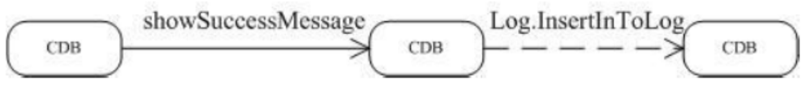


Figure 14: Aspect state model for Log

2) Aspect DataBaseAccess.

Aspect DataBaseAccess sets the database address for later operation before the system initializes an object of AddInf.

Declare AddInf (Initial, setAddress, Initial) // declare new transition in **AddInf**

pointcut ***connectAddInfor*** (Initial, jbInt, open): AddInf(Initial, jbInt, open)

advice ***connectAddInfor:***

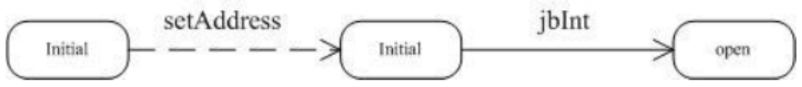


Figure 15: Aspect state model for DataBaseAccess

3) Aspect Timing.

Before opening the user interface, Aspect checks the time spent after user logging into the system. As shown in Figure 16, if the time cost since logging in is more than the restriction, Aspect will give the advice to close the window, which makes the state ends in state close.

Declare AddInf (Initial, jbInt, close) // declare new transition in **AddInf**

pointcut ***addStudent*** (Initial, jbInt, open):AddInf(Initial, jbInt, open)

advice ***addStudent:***

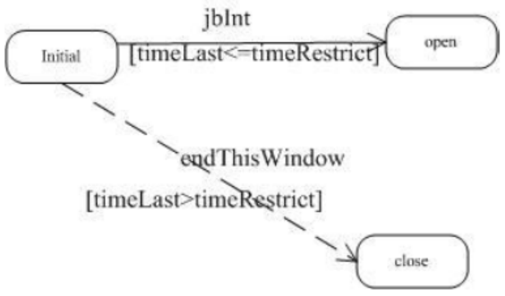


Figure 16: Aspect state model for Timing

3.2.3 Woven Model

Our woven model finds the modification due to the pointcut of Aspect models. Figure 17 shows the woven model for AddInf. Aspect log, Aspect DataBaseAccess, and Timing are woven into the base model. The dash lines point out the change of the model.

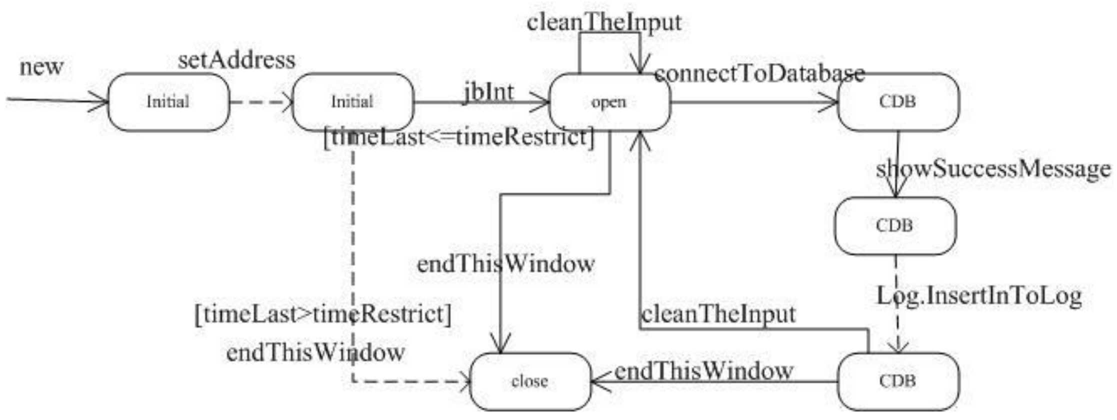


Figure 17: Woven model for AddInf

CHAPTER 4: TESTING ASPECT-ORIENTED PROGRAM

To test if the AOP is modeled and implemented correctly, we first find the scope to discuss. In this thesis, we discuss the unit tests for AOP, which is to test the individual units of AOP source code by programmers. Aspects of crosscutting concerns change the executing sequences. To catch these changes and show the logic sequences, the best way is to start from analyzing its model. Then we generate test suites from the modeling point of view. We call this approach Model-based testing. No matter what method we discuss, there are some major concepts that need to be clarified. Meanwhile, these concepts decide if the testing approach is valid and effective.

Testing criteria is the criterion that defines what constitutes an adequate test [34]. Due to the change brought by Aspect, the selection of testing criteria should be considered more on paths and statements

What objects are to be tested decides the focus of the approach such as which parts will be tested? One of our objects is to test Aspect parts, another is the whole program.

Test case generation is the exact method of the approach generating test suites from AOP codes. The cost during this process is used to evaluate the efficiency of the approach.

Tool support decides if the approach can be used widely or developed further. The approach with tool support is easier to become popular. It is also a key to evaluate the efficiency.

There are different kinds of approaches to test Aspects as mentioned in chapter 2. In this section, we introduce Flow-Based Unit Testing and the standard state-based testing.

There are two kinds of flows in program structure: control flow and data flow. Control flow is the order in which the individual statements, instructions, or function calls of an imperative or a declarative program. Data flow is the movement of data in a system, showing how the data is processed. Both control flow and data flow provide clear logic sequences we expect for program execution. Zhao's work [7] is an early approach of Aspect-Oriented program testing with the data flow which he calls Data-Flow-Based Unit Testing. His approach tests Aspects and classes in which behavior may be affected by one or more Aspects and performs intra-module testing, inter-module testing, and intra-Aspect or intra-class testing for each Aspect or class. In his approach, a control flow graph is used to compute def-use pairs. Such information is useful to guide our selection of tests.

According to Zhao's Three-level Unit Testing method, we can build the framed control flow graphs (FCFG) for Aspects and classes related to advices**.** Once we get the FCFG for the Aspect or class, we are able to use existing data flow analysis algorithms to compute the def-use pairs. Such information we get can guide the selection of tests for Aspects or classes. Class Connection, Aspect timing, and Aspect billing are components of a Telecom system. The mark before each line of the code is an indicator of flow such as s1 means statement 1, me6 means method 6, and the number is assigned according to the line's number.

By analyzing the program, we get FCFG for connection class, Timing Aspect, and Billing Aspect in Figure 18. After the connection is initialized, it calls method complete (me15). Then the state is set as complete (s16) with system printing out message (s17). After complete function is called (ae29), Aspect Timing starts to record the time (s30), which cause the returning of the fram. Thus we have flow: fram call-me15-s16-s17-ae29-s30-fram return.

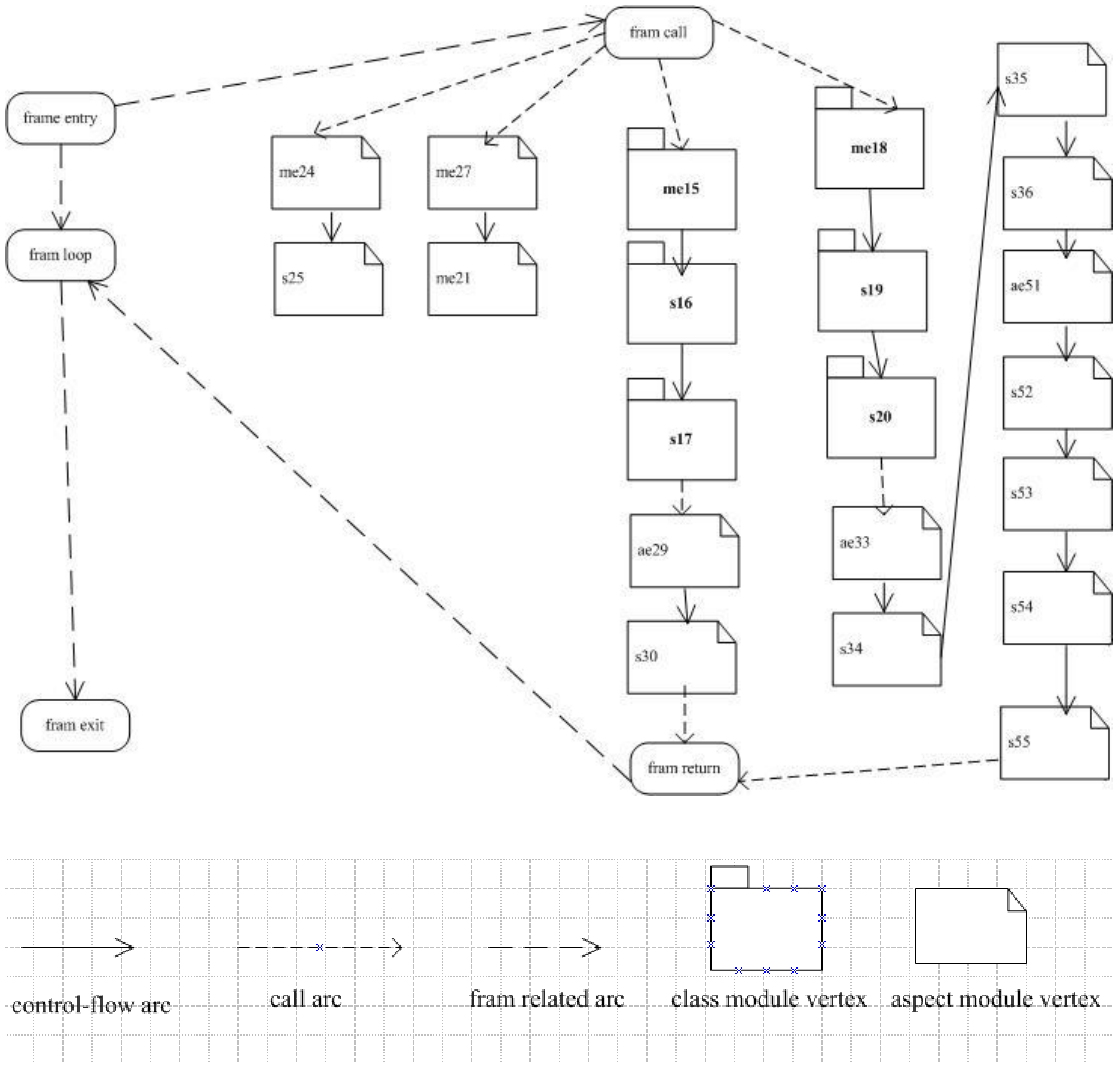


Figure 18: FCFG for connection class, Timing Aspect, and Billing Aspect

To perform the DFBUT, we need both control flow and data flow information for each class or Aspect. Zhao [7] provides the unit testing tool. His tool has three components including the driver generator, the compiler, and the test case generator. The driver generator generates test drive, which runs test cases and checks related syntax and results when the test driver is running. Currently, the test driver is generated by hand.

The compiler analyzes the program to get control flow information and data flow information. Using the information above, the compiler constructs the control flow graph. As a result, the compiler constructs the framed control flow graph for each Aspect and class. Meanwhile, it computes def-use pairs for each module.

4.1 Testing Aspects with state models: an overview

In the testing process, according to the testing paradigm shown in chapter 3, class tests are generated from the class state model and Aspect tests are generated from the woven model. The woven models are obtained by composing Aspect models into base models and both class models and woven models are represented in state machines. If base classes pass all state tests, but not Aspect tests, the failure is due to Aspects' faults. From a transition tree, we can generate test code. Test code generation from transition trees is not our focus in this paper, and more details can be found in [3]. Figure 19 shows our testing process.

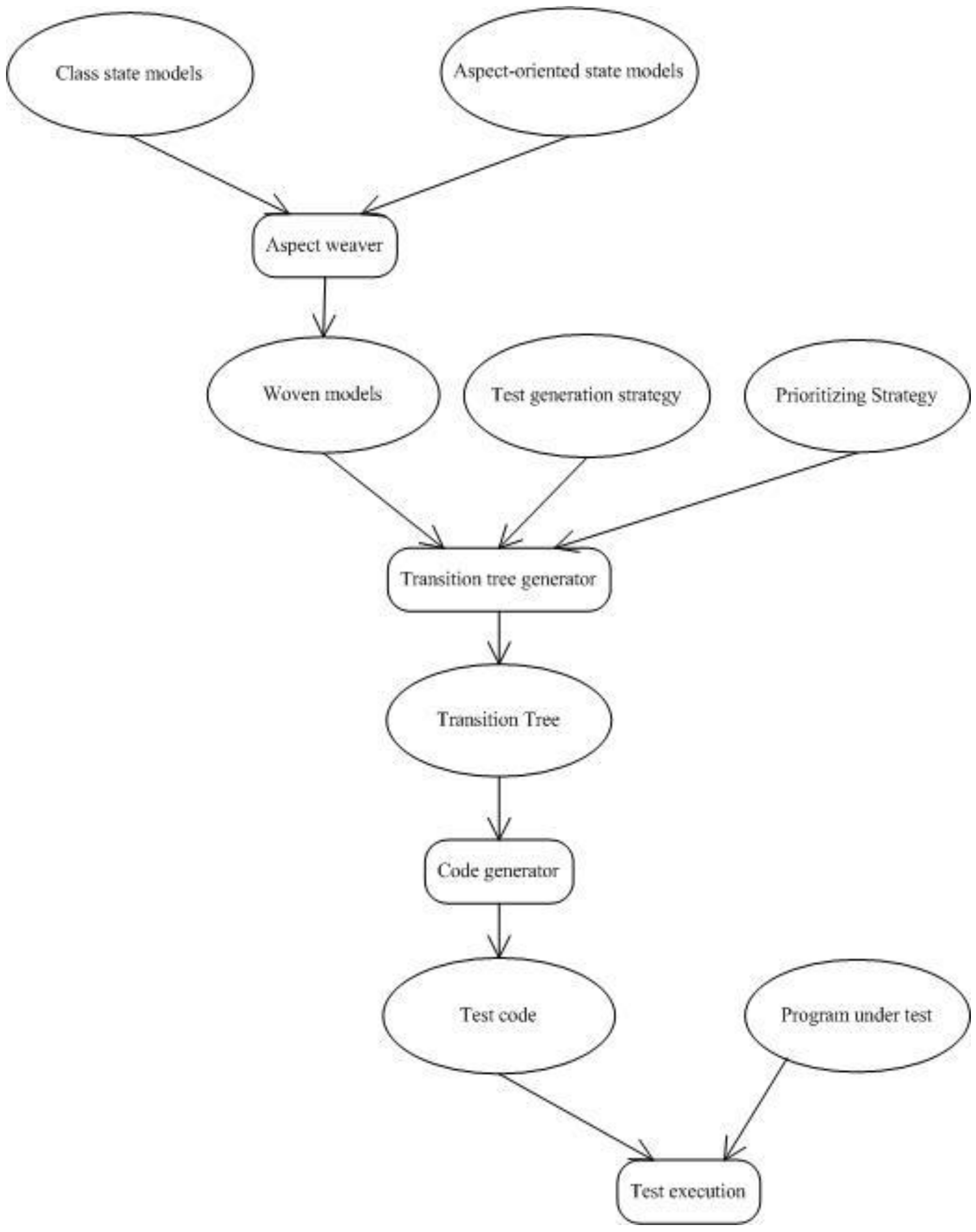


Figure 19: Overview of testing process

4.2 Generation of Transition Tree

The reason we change state model to Transition Tree (TT) is because each path of the tree (from the root to the leaf) is a test case. Nodes in the TT are states in woven model. Then we need a test strategy to make sure our test is adequate. In this paper, we discuss three different test criteria, including State Coverage (SC), Transition Coverage (TC), and Round-trip Coverage (RC).

4.2.1 State Coverage

If a test suite covers each state at least once, it achieves the state coverage. Thus we define the adequacy criterion of SC: If each state S in program P is checked by at least one test case in S, the outcome of each test execution is passed.

To generate a transition tree from the state model using state coverage, nodes will be expanded and the initial state is set as the root. If it does not arrive at the ending state and its precondition is satisfied, we create a child node for the current node, and then the state in the new node is arrived. Then we expand the new node by searching for transitions starting with the new node. We will stop until no more transition is available.

4.2.2 Round-trip Coverage

A basic round-trip test suite consists of several test sequences. The resultant object state of each sequence must occur at least once in other sequences.

4.2.3 Transition Coverage

The adequacy criterion of transition coverage is defined as follows: Each transition in the class state model must be covered. The test suite will be created according to the transitions in the state model. Each transition starting from first state to last state is defined as a test sequence.

We get TT for woven model in Figure 20. The root of TT without prioritizing is Initial state. The terminal node is open state and close state. The dash line indicates the effect of the Aspect.

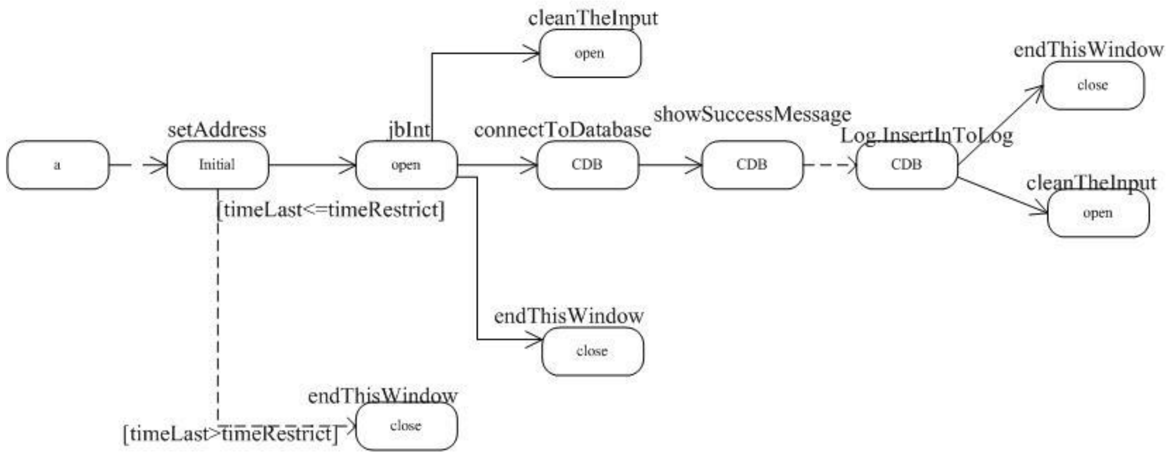


Figure 20: Transition Tree without Prioritizing

Thus we have following 5 test paths of the TT:

Path 1: a→Initial→close

Path 2: a→Initial→open→open

Path 3: a→Initial→open→close

Path 4: a→Initial→open→CDB→CDB→CDB→close

Path 5: a→Initial→open→CDB→CDB→CDB→open

According to TT, we can get 5 test cases.

4.2.4 Contrast and Problems

If three coverage criteria are compared with each other (more details can be found in the case study), RC kills the most mutants. However, the amount of test cases generating from RC is the most, which is severely time-consuming comparing with other two strategies. Although SC and TC kill fewer mutants, they are faster than RC. Can we save the time of killing the mutants by modifying the testing strategy without changing the killing rate? If the program to test is large enough, will SC and TC kill more mutants? To find answers to the above questions, we have to solve the state explosion problem, which means there are more states to test than we expected in modern software system.

4.3 Prioritizing Aspect Tests

The state explosion problem is a threat to the State-based approach, and our approach finds the path that is new in test suite. First of all, we introduce the equivalent priority value assignment for prioritizing tests. Then we introduce the non-equivalent priority value assignment.

4.3.1 Equivalent Priority value assignment

After getting the base model (as shown in Figure 14) and woven model (as shown in Figure 18), we can compute the priority value T for each transition in the woven model according to the change between the two models. The priority value is the number of components that are new in a base class model. In equivalent priority value assignments, if any of the new components are recognized, it will be assigned the same priority value 1. If none of the component is new, the priority of the transition remains 0. For example, as shown in following, five components in transition T2 after weaving are all new, and each new component is assigned with an equivalent priority value 1. After summing up, T for T2 is 5. For transition T5, its five components remain the same after weaving with Aspects. Thus its priority value is 0.

T2 (Initial, setAddress, Initial)	T=5
T3 (Initial, jbInt, open)	T=1
T5 (open, cleanTheInput, open)	T=0
T6 (open, endThisWindow, close)	T=0
T7 (open, connectToDatabase, CDB)	T=1
T8 (CDB, showSuccessMessage, CDB)	T=2
T9 (CDB, Log.InsetInToLog, CDB)	T=5
T10 (CDB, cleanTheInput, open)	T=0
T11 (CDB, endThisWindow, close)	T=0

Then we sort the T for each transition, the transitions with higher T value have greater priority to test. In this case, transitions which are changed by Aspect will be tested earlier.

4.3.2 Prioritizing Aspect Test for transition coverage

After getting the priority value for each transition, we implement the different test criteria to the test.

Test criterions are rules that define test requirements including what is an adequate test. At first, we use transition coverage. The generation algorithm for the transition coverage is shown as follows:

1) Begin.

2) Compute T value for each t.

3) At first, sort the priority value in a descending order. Choose the transition with the highest priority as the root. For each transition which has a lower priority than its previous one, create a child node of the current node if the transition is not covered yet and its precondition is satisfied. The new node represents the resultant state of the transition. The transition is marked as arrived. Keep expanding the new node.

In our example, we sort the transitions based on its priority in a descending order (starting with T1): T2, T3, T9, T8, T4, T7, T5, T6, T10, and T11.

T2, T3, and T5 have the same highest priority 5. They are three positions of the transition tree. Then the expanded node goes to T8 with priority value 2. T4 and T7 which have priority value 1 are expanded after T8. From the TT, we can get the test sequences after prioritizing as following:

Path 1: Initial→close

Path 4: Initial→open→CDB→close

Path 5: Initial→open→CDB→open

Path 2: Initial→open→open

Path 3: Initial→open→close

4.3.3 Prioritizing Aspect Test for round-trip coverage

For round-trip coverage (here we use basic round-trip coverage), the test case covers at least one round-trip path for each reachable node.

The generation algorithm for the round-trip coverage is defined as following:

1) Sort the priority value of transition in a descending order.

2) Create the initial state node and put it into a queue for expansion.

3) Search all events. Find the transition that starts with current node. If two or more transitions are found, pick the one with higher priority value. For each found transition for the given event, if its precondition is satisfied, create a child node of the current node. The new child node represents the resultant state of the transition. If the resultant state is not in the tree, expand the new node. If no transition for the given event is found or no precondition is satisfied, we create a new childe node for the event and marked it as a negative test, the priority value for negative test is 0.

4.3.4 Prioritizing Aspect Test for State Coverage

The generation algorithm for the state coverage is shown as following:

1) Sort the priority value of transition in a descending order.

2) Find the transition with a higher priority that starts with the state represented by current node. If the current node is the root, the transition of it is initializing the object.

3) If the end state is not yet reached and its precondition is satisfied, create a child node for current node, then the state in the new node is marked as reached. Then expand the new node by searching for transitions starting with the new node.

4.3.5 Non- Equivalent Priority value assignments

Equivalent priority value assignments will accelerate the failure report, now we want to improve this acceleration. One of the solutions is different weight value assignments for new components. In our approach, we assign a different weight value for each new component of the transition. We assign the ending state, the precondition, and the post-condition with value 2 if any of them is new, because they are the most significant elements changed by Aspect weaving. By this mean, preconditions, post-conditions, and ending states will have higher priority if they are found after weaving. Thus, similar changes (especially for same amount of changes) after weaving will be separated in different stages. For example, transition A and transition B both have 3 changes after weaving. Start state, event, and precondition are new in A, thus its priority value is 4(1+1+2). Precondition, event, and post condition are new in B, thus its priority value is 5(2+1+2). B has higher priority than A. B will be tested before A.

Transitions of the woven model and their T values under non-equivalent priority assignments are shown below: the priority value for T2, T3, T4, T8 and T9 are different from ones in section 4.1. The priority value for T4 is no longer 1, which makes it have higher priority than T7. Similar, T9 is no longer the same as T2 and T3. T2 and T3 will be tested before T9.

T2 (Initial, setAddress, Initial)	T=8*
T3 (Initial, endThisWindow, close)	T=8*

T4 (Initial, jbInt, open)	T=2*
T5 (open, cleanTheInput, open)	T=0
T6 (open, endThisWindow, close)	T=0
T7 (open, connectToDatabase, CDB)	T=1
T8 (CDB, showSuccessMessage, CDB)	T=4*
T9 (CDB, Log.InsetInToLog , CDB)	T=7*
T10 (CDB, cleanTheInput, open)	T=0
T11 (CDB, endThisWindow, close)	T=0

4.3.6 Test input selection

Our approach is mainly about the model level (design models and test models). We do not discuss too much about the code generation. But there is one solution we want to point out when we deal with the input values of test suites. Most test sequences in our case study do not require input values but when we need them, we can use a dynamic way to distribute the input values. In our test cases, we read input values from a XML file, which can be easily modified by testers different values for each assertion can be saved in different positions in a XML file. Therefore, we do not have to change test suites when we test different input values.

We can also use tools. For example, in Xie's work [10], they provide a tool for generating test inputs of the AspectJ program.

CHAPTER 5: CASE STUDY

To evaluate our prioritized Aspect testing, we use two case studies Telecom and SIMS. All their base classes and Aspects can be modeled by FSM.

5.1 Introduction of AspectJ projects

At first, we use a small-scale project Telecom to implement our test. Then we use a large-scale project SIMS to illustrate that prioritized Aspect testing can accelerate the failure report when there are many test sequences.

5.1.1 Telecom

Telecom is a simulation system of communication. Figure 21 shows the main architecture of Telecom. Customer, Connection, and Timer are three classes of the system. Billing, Timing, and TimerLog are three aspects. The Timing Aspect keeps a record of the connection time for customers individually. Each Connection object has a timer. The Billing Aspect adds a bill to the system. When the program is running, Aspect Billing works before Timing to enable the advice to run after Timing's advice on the same join point.

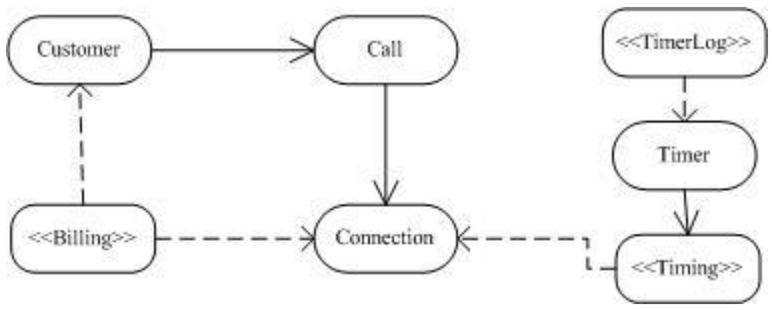


Figure 21: Architecture of Telecom

5.1.2 SIMS

SIMS (Student information management system) is a large system developed by myself to test the interaction between classes and Aspects. SIMS offers a friendly user interface and connects with Microsoft Access Database, which is a breakthrough comparing with other experiment-purpose AspectJ projects. Teachers and staffs manage students' information and courses information in the system. Meanwhile students are able to use the system to select courses and check their scores. SIMS can be further modified.

There are four main functional modules in SIMS. Main frame module is the commander that decides the movement. Student Information Management module has basic operations on student information. Course selection module manages the course and its information such as classroom, textbook, etc. Searching Score module helps students search their score corresponding to the course they are taking.

Five Aspects work as a compensation with base class models in SIMS. Aspect authorization decides which function the user cannot use. Aspect Log keeps a record of vulnerable operations. Aspect Timing manages the time a user spent since first logging into the system. If the waiting time is longer than the restriction, the user will be forced to log out. Aspect DataBaseAccess manages the database address. Thus the database address can be modified in an easy way instead of changing it manually from the code. Aspect CourseSelectLimit makes students select the right category for courses and the right amount of courses.

Figure 22 shows the main architecture of SIMS and the dash lines indicate the Aspects.

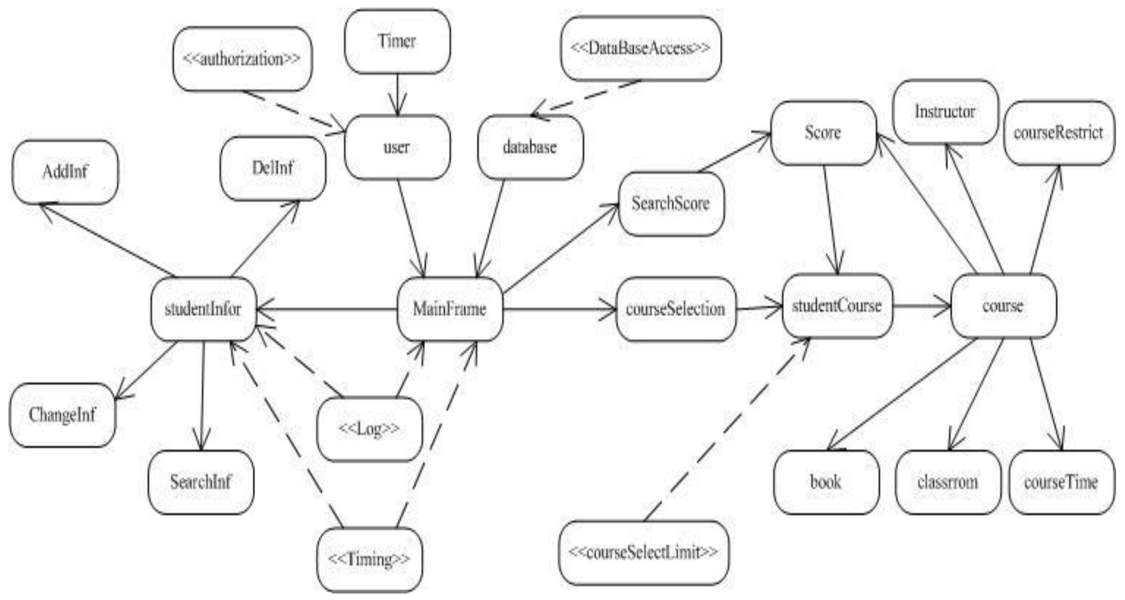


Figure 22: Architecture of SIMS

Table 2 shows different subjects of two projects. It is obvious that the scale of SIMS is larger than Telecom. Total LOC of SIMS is more than three times of Telecom's.

Subjects	**Telecom**	**SIMS**
Lines of code(LOC)	731	2794
Number of classes	10	21
LOC of classes	590	2549
Number of Aspects	3	5
LOC of Aspects	141	245

Table 2: Subjects of two projects

5.2 Evaluation phases

We evaluate our case study in two steps: The correct program is tested at first, and then we test the fault program.

From base classes, we generate state models, which is a guideline to generate tests using three different coverage strategies (transition coverage, round-trip, and state coverage).We test base class and it passes all tests without fault. Using a different coverage strategy, it generates Aspect tests from the woven state without prioritization and generates tests with prioritization in different priority value assignments. Finally, we run test suits with their base classes and they all pass the tests.

In the second step, we inject Aspect mutants and test them without prioritizing tests and with prioritizing tests, respectively. According to the Aspect mutant operators [6], we create Aspect mutants. We test each mutant with three different coverage criteria without prioritization and with prioritization. Then the priority value assignments are changed and each mutant is under the test with three different coverage criteria. Mutation operators for AspectJ are generated according to Aspect-Oriented faults [6]. There are 15 mutation operators for pointcut expression, 5 mutation operators for AspectJ declarations, and 6 mutation operators for advice definitions and implementations. In total, there are 21 mutant operators for each Aspect. In mutation operators for pointcut expression, there are 7 pointcut weakening operators, 2 pointcut strengthening operators, 3 pointcut weakening operators, and 3 pointcut changing operators.

Mutants that are generated from the operators provide a comprehensive coverage of Aspect faults, including incorrect pointcut, incorrect advice, and incorrect inter-type declaration. We created the Aspect mutants manually, search each Aspect, modify the code, and inject faults in the form of mutants.

5.3 Evaluation results

The computer used in our experiments is desktop PC: Dell Studio (Pentium Dual Core, 3.0GHz, 3GB RAM). Operating system is Windows Vista.

We get test results for two projects without prioritizing. To evaluate coverage on different projects, we use Fault Detection Rate (FDR) as the evaluation parameter. FDR is the average number of mutants killed by the test suite. FDR for prioritized and non-prioritized testing are the same, because test suites generated without and with prioritization kill the same mutants, the only difference is their test sequences.

Table 3 shows the fault detection rates of three Coverage Criteria on two different projects. TC refers to transition coverage. RT refers to round-trip. SC refers to state coverage. Round-trip kill more mutants comparing with other coverage criteria.

Subject	**Coverage Criteria**	**Tests**	**Mutants**	**Mutants Killed**	**FDR**
Telecom	TC	12	68	49	72%
	RT	22	68	62	91.2%
	SC	8	68	43	63.2%
SIMS	TC	18	120	102	85%
	RT	23	120	105	87.5%
	SC	14	120	88	73.3%

Table 3: FDR without prioritization

We use two factors to evaluate the improvement of test prioritization. The first factor is the average amount of tests it ran to find the first mutant, which we call ATFM in short. ATFM' represents the average amount of tests ran to find the first mutants with prioritization. The second factor is the average execution time for killing the mutants, which we call ATIM in short. The unit of ATIM is second. ATIM' represents the average execution time for killing the mutants with prioritization. To get the improvement of ATFM, we use following formula: Improvement = (ATFM-ATFM')/ATFM. Similarly, we get the Improvement of ATIM using formula: Improvement = (ATIM-ATIM')/ATIM. Table 4 and Table 5 show the improvement of ATFM and ATIM using the Equivalent priority assignment. As shown in table 4, the improvement of ATFM is more obvious on a large-scale project than on a small-scale project. While as shown in table 5, the improvement of ATIM is more obvious on small-scale project than in the large size project.

Subject	**Coverage Criteria**	**ATFM**	**ATFM'**	**Improvement**
Telecom	TC	1.8	1.6	11.1%
	RT	2.9	2.5	13.7%
	SC	1.4	1.2	14.29%
SIMS	TC	2.33	1.91	18%
	RT	3.2	2.5	21.88%
	SC	1.82	1.45	20.3%

Table 4: ATFM improvement- Equivalent priority assignment

Subject	**Coverage Criteria**	**ATIM**	**ATIM'**	**Improvement**
Telecom	TC	0.28	0.24	14.29%
	RT	0.55	0.4	27.27%
	SC	0.36	0.28	22.22%
SIMS	TC	0.72	0.61	15.27%
	RT	0.93	0.73	21.5%
	SC	0.88	0.78	11.36%

Table 5: ATIM improvement- Equivalent priority assignment

Table 6 and Table 7 show the improvement of ATFM and ATIM using the non-Equivalent priority assignment. The Improvement values in Table 6 and Table 7 are greater than the corresponding ones in Table 4 and Table 5.

Subject	**Coverage Criteria**	**ATFM**	**ATFM'**	**Improvement**
Telecom	TC	2.13	1.79	16.1%
	RT	3.2	2.47	23.2%
	SC	3.66	2.78	24.4%
SIMS	TC	2.33	1.8	22.1%
	RT	4.5	3.12	30.6%
	SC	2.8	2.13	23.9%

Table 6: ATFM improvement- non-Equivalent priority assignment

Subject	**Coverage Criteria**	**ATIM**	**ATIM'**	**Improvement**
Telecom	TC	0.32	0.23	28.1%
	RT	0.65	0.37	43%
	SC	0.37	0.28	24.3%
SIMS	TC	0.73	0.57	21.9%
	RT	0.91	0.62	31.9%
	SC	0.88	0.7	20.5%

Table 7: ATIM improvement- non-Equivalent priority assignment

According to Table 4, Table 5, Table 6, and Table 7, we can calculate the improvement of different priority assignments on ATFM and ATIM. To find this improvement, we use improvement from Table 6 minus the improvement from Table 4, and use improvement from Table 7 minus the improvement from Table 5. The improvement from the Equivalent priority assignment to the non-Equivalent priority assignment is shown in Table 8. The result shows that the non-Equivalent priority assignment will improve the prioritized testing. In all, the improvement on ATIM is greater than on ATFM.

Subject	**Coverage Criteria**	**Improvement on ATFM**	**Improvement on ATIM**
Telecom	TC	5%	13.81%
	RT	9.5%	15.73%
	SC	10.11%	2.08%
SIMS	TC	4.1%	6.63%
	RT	8.72%	10.4%
	SC	3.6%	9.14%

Table 8: Improvement of different priority assignments

Both case studies demonstrate that Aspect testing with prioritization improves test execution performance. Because Aspects weaving into base class are incremental modifications, the Aspect faults are generated in this modification. When we test the modification first, the fault can be found immediately. Meanwhile, a decent priority assignment strategy can improve the prioritization of testing. A better priority assignment will save more time.

CHAPTER 6: CONCLUSION

This paper presents a way of Aspect-Oriented Modeling and testing. At the beginning, we introduce what are Aspect modeling and testing. Then we use a finite state machine to build state models for the Aspect-Oriented system. The state model is tested with the transition coverage, the round-trip coverage, and the state coverage. After getting the woven model, we test it without prioritizing and with prioritizing strategy. The prioritizing strategy is implemented by identifying new elements brought into state models, and the new transition tree from the prioritizing testing is used to generate the JUnit test cases for testing the AOP program. Different testing coverage criteria and a dynamic prioritizing strategy have been implemented in the prioritizing testing. In order to investigate the advantages of prioritizing tests, we performed case studies of several AOP programs, which have been further tested by mutation testing to confirm the effectiveness of the approach.

In future work, we will focus on implementing the prioritization test process automatically, which includes state models generation, Aspect model generation, woven model generation, and test cases generation. We also want to find the optimist algorithm for the priority assignment that will fit different coverage criteria and find the best combination for priority values. It will have a significant effect on testing large-scale projects.

REFERENCES

[1] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J. M. Loingtier, and J. Irwin, “Aspect-Oriented Programming”, In Proc. of ECOOP, LNCS 1241, pp. 220-242, 1997.

[2] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W.G. Griswold, “ An Overview of AspectJ”, In Proc. of ECOOP, pp. 327-353, 2001.

[3] D. Xu, W. Xu, and W. E. Wong, “Automated Test Code Generation from Class State Models”, International Journal of Software Engineering and Knowledge Engineering, pp. 599-623, 2009.

[4] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B.M. Horowitz, “Model-Based Testing in Practice”, In Proc. of the International Conf. Software, 1999.

[5] D. Xu, and W. Xu, “State-Based Incremental Testing of Aspect-Oriented Programs”, In Proc. of the International Conf. on Aspect-Oriented Software Development, pp. 180-189, 2006.

[6] C. Ferrari, J.C. Maldonado, and A. Rashid, “Mutation Testing for Aspect-Oriented Programs”, In Proc. of the First International Conf. on Software Testing, Verification, and Validation, pp.52-61, 2008.

[7] J. Zhao, “Data-Flow-Based Unit Testing of Aspect-Oriented Programs”, In Proc of the 27th Annual IEEE International Computer Software and Applications Conference, pp.188-197, 2003.

[8] Y. Zhou, D. Richardson, and H. Ziv, “Towards a Practical Approach to Test Aspect-Oriented Software”, In Proc. of the 2004 Workshop on Testing Component-Based Systems, 2004.

[9] O. A. L. Lemos, J. C. Maldonado, and P. C. Masiero, “Structural unit testing of Aspect programs”, In the 1st Workshop on Testing Aspect-Oriented Programs, 4th International Conf. on Aspect-Oriented Software Development, Chicago, Illinois, 2005.

[10] T. Xie, and J. Zhao, "A Framework and Tool Supports for Generating Test Inputs of AspectJ Programs", In Proc. of the International Conf. on Aspect-Oriented Software Development, pp. 190-201, 2006.

[11]G. Xu, Z. Yang, H. Huang, Q. Chen, L. Chen, and F. Xu, "Jaout: Automated generation of Aspect-Oriented unit tests", In Proc. of the 11th Asia-Pacific Software Engineering Conference, Busan, South Korea, 2004.

[12] H. Rajan, and K. Sullivan, "Generalizing AOP for Aspect-Oriented Testing", In Proc. of the Fourth International Conference on Aspect-Oriented Software Development, pp.14-18, 2005.

[13] A. Deursen, M. Martin, and L. Mooned, "A systematic Aspect-Oriented Refactoring and Testing Strategy, and its Application to JhotDraw", In Proc. of the 2005 ICSE Workshop on Modeling and analysis of concerns in software, St. Louis, Missouri, 2005.

[14] J. Zhao, T. Xie, and N. Li, "Towards regression test selection for Aspectj programs", In the 2nd Workshop on Testing Aspect-Oriented Programs, International Symposium on Software Testing and Analysis, Portland, Maine, 2006.

[15] R. T. Alexander, J. M. Bieman, and A. A. Andrews, "Towards the systematic testing of Aspect-Oriented programs", Technical Report CS-4-105, Department of Computer Science, Colorado State University, Fort Collins, Colorado, 2004.

[16] P. Anbalagan, and T. Xie, "Efficient Mutant Generation for Mutation Testing of Pointcuts in Aspect-Oriented Programs", Second Workshop on Mutation Analysis, Los Alamitos, CA, USA, 2006.

[17] M. Mortensen, and R. T. Alexander, "An approach for adequate testing of Aspectj programs", In the 1st Workshop on Testing Aspect Oriented Programs, 4th International

Conference on Aspect-Oriented Software Development, Chicago, Illinois, 2005.

[18] D. Stein, S.T. Hanenberg, and R. Unland, "Designing Aspect-Oriented Crosscutting in UML", AOSD-UML Workshop, nschede, the Netherlands, 2002.

[19] G. Zhang, and M. Hölzl, "HiLA: High-Level Aspects for UML State Machines", In Proc. of the 14th International Aspect-Oriented Modeling, 2009.

[20] A. Jackson, J. Klein, and B. Baudry, "Executable Aspect Oriented Models for Improved Model Testing", ECMDA workshop on Integration of Model Driven Development and Model Driven Testing, 2006.

[21] W. Xu, and D. Xu, "A model-based approach to test generation for Aspect-Oriented programs", In the 1st Workshop on Testing Aspect Oriented Programs, 4th International Conference on Aspect-Oriented Software Development, 2005.

[22] D. Xu, O. El. Ariss, and W. Xu, "Aspect-Oriented Modeling and Verification with Finite State Machines", Journal of Computer Science and Technology, pp. 949-961, 2009.

[23] W. Xu, and D. Xu, "State-based testing of integration Aspects", In Workshop on Testing Aspect-Oriented Programs, International Symposium on Software Testing and Analysis, Portland, Maine, 2006.

[24] W. Xu, D. Xu, V. Goel, and K. Nygard, "Aspect flow graph for testing Aspect-Oriented programs", Journal of Systems and Software, Volume 80, pp.862-882, 2006.

[25] D. Xu, and J. Ding, "Prioritizing State-Based Aspect Tests", 2010 Third International Conference on Software Testing, Verification and Validation, pp.265-274, 2010.

[26] O. A. L. Lemos, and C. V. Lopes, "Testing Aspect-Oriented programming pointcut

descriptors”, In the 2nd Workshop on Testing Aspect-Oriented Programs, International Symposium on Software Testing and Analysis, Portland, Maine, 2006.

[27] P. Anbalagan, and T. Xie, “Apte: Automated pointcut testing for Aspectj programs”, In the 2nd Workshop on Testing Aspect-Oriented Programs, International Symposium on Software Testing and Analysis, Portland, Maine, 2006.

[28] R. B. France, I. Ray, G. Georg, and S. Ghosh, “An Aspect-Oriented approach to design modeling”, IEEE Proceedings – Software, Special Issue on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design, 2004.

[29] P. Sa ́nchez, L. Fuentes, A. Jackson, and S. Clarke, “Aspects at the right time”, LNCS Transactions on Aspect-Oriented Software Development IV, Special Issue on Early Aspects, LNCS 4640, pp. 54–113, 2007.

[30] J. Klein, F. Fleurey, and J.M. J éz équel, “Weaving multiple Aspects in sequence diagrams”, Transactions on Aspect-Oriented Software Development, LNCS 4620, pp.167-199, 2007.

[31] G. Mussbacher, D. Amyot，and M. Weiss, “Visualizing Early Aspects with Use Case Maps”, Transactions on Aspect-Oriented Software Development III, LNCS 4620, pp.105-143, 2007.

[32] E. Baniassad, and S. Clarke, “Theme: An Approach for Aspect-Oriented Analysis and Design”, In Proc. of 26th International Conf. Software Eng. IEEE CS Press, pp.158–167, 2004.

[33] D. Stein, S. Hanenberg, and R. Unland, “A UML-based Aspect-Oriented Design Notation for AspectJ”, In Proc. of the 1st International Conf. on Aspect-Oriented software development, Enschede, The Netherlands, 2002.

[34] H. Zhu, P. A. V. Hall, and J. H. R. May, “Software unit test coverage and adequacy”, ACM,

Survey, pp.366–427, 1997.