Towards A Practical Approach to Test Aspect-Oriented Software

Yuewei Zhou, Hadar Ziv, Debra Richardson

Department of Informatics
School of Information and Computer Science
University of California, Irvine
{yueweiz, ziv, djr}@ics.uci.edu

Abstract: Aspect-Oriented Programming (AOP) provides new constructs and tools to handle cross-cutting concerns in programs. Fully realizing the potentials of Aspect-Oriented Software Development requires new abstractions and techniques for testing. This paper proposes a first step towards a practical approach to test aspect-oriented software. The proposed approach is accompanied by a selection algorithm that can select test cases that are relevant to aspects under test. A new testing coverage definition is proposed to specify the sufficiency of test cases on the aspect being tested. A tool is developed to support the approach, automating test case selection and coverage calculation. A detailed case study of banking account processing illustrates this initial approach.

1 Introduction

Aspect-Oriented Programming (AOP) [Ki97] was proposed to address the issue of separating cross-cutting concerns, such as logging, security, transaction, and reliability, from a system's main application logic. In traditional procedural or Object-Oriented Programming (OOP), handling of functional and non-functional requirements (also called extra functional requirements) is intertwined. Developers write programs using abstraction constructs like procedures and classes. Each such construct typically addresses both functionality and non-functional properties. AOP explicitly provides constructs to separate cross-cutting non-functional concerns from the application logic. It allows programmers to develop application logic and non-functional properties separately, and it enables programmers to weave separately developed logic and properties together using automated support. This technique advances the traditional pursuit of modular programming and provides developers with a new effective means for software construction.

The most well known AOP programming system is AspectJ [As04, La03]. AspectJ is an aspect-oriented extension to Java. It provides new constructs to better handle cross cutting concerns. In addition to the concept of class (which we call regular class, or just class), the unique key concepts of AspectJ are point cuts, advice, and aspects. Point cuts define where concerns should be handled. Each such individual place is called a join point. Advice defines how concerns should be handled. An aspect is an abstraction that encapsulates all handling related to a concern. The AspectJ compiler takes a set of AspectJ programs and outputs Java byte code that can execute on a standard Java Virtual Machine.

AOP has evolved from its original form into a more comprehensive methodology, Aspect-Oriented Software Development. Due to the relative immaturity of aspect-oriented software development, it is not surprising that many topics have not yet been explored to the extent necessary to fully realize the potentials of this promising paradigm. For instance, effective testing techniques for aspect-oriented software remain to be invented and developed. Testing of AOP software is essential to ensure quality, yet as we know, testing is relatively more expensive than the earlier phases of software development. Finding cost-effective testing techniques that address aspect-oriented software is an essential step towards a practical full-fledged aspect-oriented development paradigm.

1.1 Challenge

While Aspect-Oriented Programming eases development and maintenance, it imposes new challenges for testing and analysis. Aspects make explicit where and how a concern is addressed, in the form of joint points and advice. In static program text, the specification for concern handling is concentrated in one place, the aspect. However, dynamic execution of advice happens around all join points in classes as defined by point cuts. An aspect can define a piece of advice that is executed at several joint points of a point cut. These join points can belong to different methods, all of which can come from different classes. The gap between static text and dynamic execution imposes a new challenge for testing and analysis.

The challenges come not only from the gap between static text and dynamic execution, but also from the characteristics of the advice relationship between aspects and classes. Aspects are concepts for handling cross-cutting concerns. An aspect can affect many classes. If we assume all regular classes form one dimension of abstraction, because they handle normal application logic, then we can argue aspects form a second dimension of abstraction that is orthogonal to the first dimension, because each of them handles one concern, and they are of different characteristics than regular classes.

Along the first dimension, there exist data and control dependence between classes. One major form of the dependence is the calling relationship between a calling method of one class to a called method of another class. Much work on testing utilizes the calling relationship [Ba98].

Aspects of the second dimension have different relationships. Since each aspect addresses a single cross-cutting concern, in the extreme case when concerns are independent, aspects will have no interactions between each other.

When aspects advise classes, the two dimensions interact. The previous discussion suggests advice of an aspect is called by methods of regular classes. However, this calling relationship is different than a traditional one between regular classes. First, in a traditional calling relationship, a caller specifies what callee it invokes and when the invocation happens. Aspects, whose advice is called, define what regular classes can invoke the advice and when the advice is invoked, through point cuts associated with the advice. Essentially it is the callee that specifies who can call and when to call. Second, the caller and callee in traditional OOP are both classes. In AOP the calling relationship happens between classes and aspects, which are different types of concepts.

Because of the similarity and difference between the traditional calling relationship among regular classes and the advice relationship among aspects and classes, we argue traditional calling-based techniques are not directly applicable, but they can be adapted to test aspect-oriented software.

1.2 Organization of Paper

The paper is organized as follows. In Section 2, we describe an approach that can be a first step towards effective testing of aspect-oriented programs. The approach is accompanied by a selection algorithm that can select test cases that are relevant to aspects under consideration. A new testing coverage definition is proposed to specify the sufficiency of test cases on the aspect under test. We develop a tool to support our approach, automating test case selection and coverage calculation. Section 3 illustrates the approach by a case study that comes from the domain of banking account processing. Related work is surveyed in Section 4. Section 5 concludes the paper and outlines future work.

2 An Initial Approach for Testing

In this section, we introduce an approach to test aspects in aspect-oriented programs. We first give an overview of our approach. Then, we outline an algorithm that can reuse test cases to reduce the cost of aspect testing. We also propose a new coverage definition for aspects. This section ends with a description of an automated tool we developed to support our approach. What is proposed here is only a first step towards a comprehensive approach for testing aspect-oriented software. This area is still full of research questions. Some possibilities for further investigation along this first proposal are discussed in the last section.

2.1 Overview

Our approach deals with unit testing, integration testing, and system testing of aspect-oriented software. It consists of four steps, shown in Figure 2.1. This approach for testing aspects in aspect-oriented software adapts standard testing processes to address characteristic issues of aspects, applying a new selection algorithm for selecting test cases, and proposing a new definition for testing coverage.

In the first step, we develop regular classes and test them. The purpose of developing and testing regular classes before testing aspects is to isolate and eliminate errors that are not aspect-related. If we can attain high confidence in the correctness of regular classes, we can have a better chance of detecting aspect-related errors. If aspects are well separated, it is possible a standalone system that only handles application logic can be developed and tested without consideration of aspects. We can apply standard techniques in this step [Be90]. Test cases developed in this step are saved for possible future reuse.

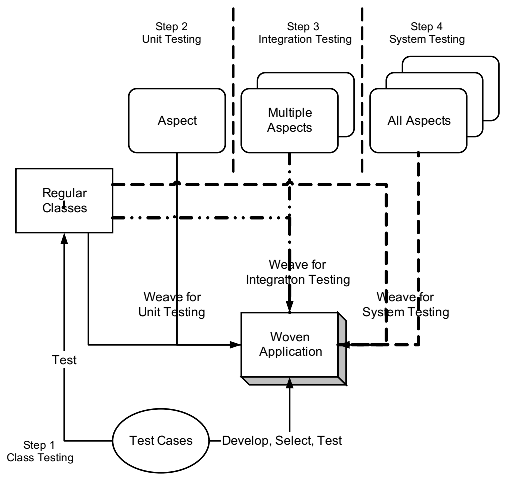

Figure 2.1, Approach for Testing Aspects

In the second step, each aspect is separately woven with regular classes, and each resulting woven application is tested to verify that the single aspect under consideration behaves as required. This step corresponds to traditional unit testing. Because each aspect is supposed to handle a single cross-cutting concern, and concerns are generally independent of each other, it is possible that we can test a single aspect without any other aspects.

Because of the advice relationship, regular classes will invoke advice from aspects, along with application logic. Because test cases lead to execution of regular classes, they also lead to execution of advice from aspects. It is possible to reuse test cases from the previous step. New test cases are developed when necessary.

During the third step of our approach, multiple aspects are woven together with classes to form a more complex application, and this application goes under testing. This step performs integration testing on several aspects. The purpose of this step is to test the interaction among aspects, and identify and eliminate errors that result from the presence of multiple aspects. Incremental testing is preferred in this step. Aspects can be classified into multiple increments. Each increment only contains a few aspects. For each woven application that incorporates an increment, only a limited amount of interaction between the aspects in the increment can lead to integration errors.

The order of integration is important to traditional integration testing. Strategies like bottom-up or top-down are used, and drivers and stubs are developed to support incremental integration testing. However, due to the nature of aspects, the order of integration might not be as important. Each aspect addresses one cross cutting concern. Most likely an aspect is independent of other aspects.

In the final step of our approach, all aspects are woven together with regular classes to form the complete system, and this system is tested. This step corresponds to traditional system testing. This step assures that the system still performs as expected when all classes and aspects are finally assembled together. The final step needs to develop new test cases, in addition to reusing test cases from previous steps.

2.2 Select Relevant Test Cases

An important element of our approach is to identify test cases relevant to an aspect under test. More relevant test cases can be developed to perform more tests on important aspects, and execution of irrelevant test cases can be avoided to reduce the cost associated with testing. We define the meaning of an aspect relevant test case as follows. We also say a test case touches an aspect when the test case is relevant to the aspect.

Definition 1 (Aspect Relevant Test Cases). If the execution of the test case **t** results in the execution of any advice of the aspect **A**, then the test case is relevant to the aspect, or **t** is **A**-relevant.

We propose an algorithm that can identify a set of test cases relevant to an aspect under test. The algorithm is listed in Figure 2.2. The input for the algorithm is a set of test cases and the aspect under test. The output of the algorithm is the subset of the test cases that are relevant to the aspect. In the first step of the algorithm, we build a relation, $\mathscr{A}$, that captures the "advise" relationship from aspects to classes. The domain of the relation is advice of aspects. The range of the relation is methods of classes. We analyze the specification of advice in aspects and build this relation. For each advice **a** of each aspect **A**, we check each method **m** of each class **C**. If the advice advises the method, we record the pair (**A.a, C.m**) in the relation $\mathscr{A}$. In the second step of the algorithm, we build a relation, $\mathscr{T}$, which captures the relationship from test cases to methods of classes that the test case executes. The domain of the relation is the available test cases. The range of the relation is the methods of the classes. For each test case **t**, we can identify each method **m** for a class **C** that it invokes. This can be accomplished with standard call graph construction and analysis [GC01]. We put each pair (**t, C.m**) into the relation $\mathscr{T}$. After constructing the two base relations, we construct an auxiliary relation, $\mathscr{AB}$, which describes the "advised by" relationship between a method of a class and an advice of an aspect. This is the inverse of relation $\mathscr{A}$. The domain of this relation is the methods of the classes. The image of this relation is the advice of the aspects. If (**A.a, C.m**) is an element of $\mathscr{A}$, then (**C.m, A.a**) is an element of the relation $\mathscr{AB}$. For each member of the input set of test cases, we find all methods that it executes, using relation $\mathscr{T}$. For each method thus found, we find the aspects that advise the method, using relation $\mathscr{AB}$. If the advising aspects contain the aspect under test, we add the test case to the result set of relevant test cases. At the end of the algorithm, the result set contains all test cases in the input set that are relevant to the aspect under test. The complexity of the algorithm is *O(lmn)*, where *l* is the number of the advice of all aspects, *m* is the number of the methods of all classes, and *n* is the number of test cases. This result is apparent from the algorithm.

2.3 Define Test Coverage

Another important consideration of testing is the sufficiency of tests for the program under test. Test cases should test enough portions of a program to establish confidence in the program. The sufficiency is defined by a coverage criterion. Many definitions of coverage have been proposed for control flow and data flow testing techniques [Cl89, RW85]. For example, if a set of test cases execute each statement of a program at least once, then the test cases meet the statement coverage criterion.

We define a test coverage criterion for testing aspect-oriented programs. A set of test cases covers an aspect under test if all methods of classes that are advised by any advice of the aspect are tested by the one of the test cases. Formally, let **A** be the aspect, let **a** be an advice of the aspect **A**, let **m** be a method, let **M** be the set of all methods from all classes, let **A(a, m)** designate **a** advises **m**, let **T** be the set of test cases, let **t** be one test case of **T**, and let **I(t, m)** designate **t** invokes **m**.

Definition 2 (Aspect Covering Tests). Test case set T **covers** Aspect A if $\forall a \in A\ \forall m \in M$, such that $A(a, m)\ \exists t \in T$, such that $I(t, m)$.

If a set of test cases does not fully cover the aspect under test, we can calculate the percentage of the coverage, based on the percentage of the advised methods that are tested by test cases. Intuitively, the percentage is the portion of advised methods that are tested by all advised methods. The formal definition is as follows. We use the same notations from above and let **|S|** be the size of set **S**.

Definition 3 (Percentage of Coverage). The percentage of coverage for a set T over an aspect A is $|\{m \in M \mid \exists\ t \in T,\ I(t, m) \wedge \exists\ a \in A,\ A(a, m)\}| \,/\, |\{m \in M \mid \exists\ a \in A,\ A(a, m)\}|$.

The algorithm proposed above can be easily extended to decide whether a set of test cases covers the aspect under test, and if not, what the percentage of coverage would be, and what advice is not tested by the test cases. The extended algorithm is given in Figure 2.3.

The first and second step is the same as in algorithm 1. In the third step, we construct a different auxiliary relation. The auxiliary relation, $\mathcal{TB}$, captures the "tested by" relationship between methods of classes and test cases. It is the inverse of relation $\mathcal{T}$. Its domain is the methods of classes, and its range is the test cases. If (t, C.m) is an element of relation $\mathcal{T}$, then (C.m, t) is an element of relation $\mathcal{TB}$.

For each advice of the aspect, we find the methods it advises, using relation $\mathcal{A}$. For each such advised method, we find the test cases that test it, using relation $\mathcal{TB}$. If there are no such test cases, or the test cases are not fully contained in the input set of test cases, then the advised method is not covered by the input set, and neither is the advice.

At the end of the algorithm, we calculate the percentage of the number of tested advised methods and the number of total advised methods. We return the percentage. One hundred percent indicates full coverage of the test set over the aspect. We also return advice that is not fully tested.

```
Algorithm 1:
SelectRelevantTestCase
Input: T: set of test cases, AUT: aspect under test
Output: {t | t ∈ T ∧ t is AUT-relevant}
Begin
    { 1. Build advice relation. }
    $\mathcal{A} := \emptyset$
    foreach aspect A
       foreach a∈A
          foreach m∈C
              if (A.a advises C.m) then
                 $\mathcal{A} := \mathcal{A} \cup$ (A.a, C.m)
    { 2. Build invoke relation }
    $\mathcal{T} := \emptyset$
    foreach t ∈ T
       foreach m∈C
          if (t invokes C.m) then
             $\mathcal{T} := \mathcal{T} \cup$ (t, C.m)
    { 3. Build auxiliary relation }
    $\mathcal{AB} := \mathcal{A}^{-1}$
    { 4. Collect relevant test cases. }
    $\mathcal{TR} := \emptyset$
    foreach t ∈ T
       foreach (t, C.m) ∈ $\mathcal{T}$
          foreach (C.m, A.a) ∈ $\mathcal{AB}$
             if A = AUT then
                $\mathcal{TR} := \mathcal{TR} \cup$ {t}
    return $\mathcal{TR}$
end
```

Figure 2.2, Select Relevant Test Cases

```
Algorithm 2:
ComputeTestCoverage
Input: T: set of test cases, AUT: aspect under test
Output: percentage of coverage, and uncovered advice
Begin
    { 1. Build advice relation $\mathcal{A}$. }
    { 2. Build invoke relation $\mathcal{T}$}
    { 3. Build auxiliary relation }
    $\mathcal{TB} := \mathcal{T}^{-1}$
    { 4. Calculate percentage. }
    $\mathcal{UC} := \emptyset$
    total := 0;
    untested := 0;
    foreach a ∈ AUT
       foreach (AUT.a, C.m) ∈ $\mathcal{A}$
          total := total + 1
          foreach (C.m, t) ∈ $\mathcal{TB}$
             if t ∉ T then
                untested := untested + 1
                $\mathcal{UC} := \mathcal{UC} \cup$ {a}
    return (total – untested)/total, $\mathcal{UC}$
end
```

Figure 2.3, Compute Test Coverage

Using this information, the tester can develop new test cases that test the yet-to-be tested advice of an aspect. The tester can also compare test cases based on their coverage of aspects. If important aspects are to be tested more thoroughly, the tester can develop several different sets of test cases that each fully cover the aspect, and apply all of them on the important aspect.

2.4 Tool Support

We developed a Java tool to help automate our approach. The tool implements the test selection and coverage calculation algorithms. It can be used to select and execute aspect relevant test cases. The tool is developed with the Eclipse environment [Ec04]. Eclipse is an open source Java integrated development environment. Its plug-in architecture enables third parties to integrate their techniques with a full-featured development environment. The AspectJ toolset comes in two forms [As04]. A standalone tool set contains a compiler (ajc) that can weave classes and aspects together into executable byte code and an aspect browser that displays advice relationships between classes and aspects. Plug-ins for popular Java development are also available. We use the Eclipse plug-in.

The plug-in provides programmatic access to the result of weaving. The access provides interfaces to retrieve the advice relationship between classes and aspects. We use this support to build the "advise" relation $\mathcal{A}$ and the "advised by" relation $\mathcal{AB}$ of our algorithms. To retain the conceptual clarity of classes and aspects, we need access to their source code. Byte code of woven applications, which mix classes and aspects together, is not suitable for our purpose.

To build the "test" relation $\mathcal{T}$ and "tested by" relation $\mathcal{TB}$, we can use any implementations of standard call graph construction and analysis. We choose another open source Java library, Barat [Ba04], for its ease of use and its capability to handle byte code. It enables us to analyze the byte code of test cases and retrieve methods of classes that these test cases invoke.

We choose JUnit as the testing framework to write and execute test cases [Ju04]. A testing framework provides basic automatic support for writing test cases, executing them, and determining whether the test cases reveal an error. JUnit is arguably the most popular such framework. We choose it mainly to improve the appeal of our approach to practical developers.

A screenshot of our tool is provided in Figure 2.4. It looks similar to the Swing TestRunner of JUnit, which is a GUI interface to execute tests and re-run failed cases.

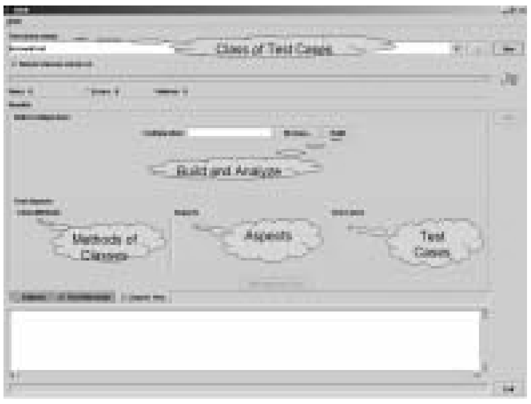

Figure 2.4, Aspect Test Runner

After receiving the input for a build configuration file, which lists classes and aspects, and a class for test cases, the tool uses the AspectJ plug-in to weave classes and aspects, and uses Barat to analyze classes and test cases. The tool uses the result of building and analysis to populate the three panels, which list methods of classes, aspects currently under test, and test cases. When the tester selects an aspect, the tool identifies the methods of classes that this aspect advises, and relevant test cases for this aspect. The tester can also select a set of test cases, and the tool identifies the methods tested by the cases and calculates the coverage for an aspect. If the test cases fully cover an aspect, the tool notifies the tester.

After finishing selection of test cases, the tester can use the tool to execute the selected test cases. The tool checks whether the execution detects any errors in the aspect-oriented software.

3 Case Study

In this section, we use a case study from the domain of banking account processing to illustrate our approach, as shown in Figure 3.1. We demonstrate how our approach can effectively test aspect-oriented software in practical situations. The example system, adapted from samples in [La03], consists of 2 classes, 2 aspects, and 5 test cases.

3.1 Classes

The first class is an Account Class. It maintains its account number and its balance in a database. The methods of this class enable getting the account number, getting and setting the balance, crediting the account, and debiting the account. When debiting the account, if the balance is insufficient for the operation, an exception is thrown.

The second class, Inter Account Transfer System, contains only one method. The method takes two accounts, debits an amount from one account, and credits the same amount to the other account.

3.2 Aspects

In this section, we use two aspects: secure access and transaction integrity. Actually there are many more concerns potentially relevant to this simple application, such as UI concern, but here we are only focusing on two.

The first aspect is an authentication aspect. Since banking account processing is a domain that requires high security, naturally some methods need authentication before they can be executed. This naturally forms a cross-cutting concern, and aspect provides an appropriate abstraction to handle the concern.

The authentication aspect defines one point cut. The `authOperations` point cut defines the join points that are subject to authentication. Its first element, `execution(public * Account.*(..))`, includes join points of the execution of any public methods of the class `Account` that takes any type of arguments and returns any type of value. The word `execution` specifies that the type of the join points is the execution of a method. The signature in parentheses specifies the access, name, parameters, and return value the method should have. Special wild card characters can be used to include multiple values. The first star (`*`) signifies that any type of return value matches the signature. The second star allows any name for the method name part. The double dots (`..`) permit any number of parameters of any type. The second element

```
public aspect AuthAspect {
  protected pointcut authOperations()
      :(execution(public * Account.*(..)))
      && !(
 execution(public * Account.credit(..))))
      .. ..

  before() : authOperations() {
      try {
        authenticate();
      } catch (LoginException ex) {
.. ..
public aspect TransactionAspect {
  protected pointcut transOperation()
    : execution(* Account.debit(..))
    || execution(*
InterAccountTransferSystem.transfer(..));

  Object around(): transactedOperation(){
    try {
      operationResult = proceed();
      _connection.commit();
    } catch (Exception ex) {
      _connection.rollback();
      throw new TransactionException(ex);
    }
.. ..
```

Figure 3.1, Aspects in Banking Account System

of the `authOperations` point cut defines another join point. It is also a method execution point. This time it has a concrete method name in the signature, so only methods of `Account` class that have the name `credit` match this join point. The join points are connected together with common logical operators. The effect here is to exclude the second, more specific join point from the first, very broad join point.

The authentication aspect defines one advice. It is a `before` advice that is associated with `authOperations`. The body of the advice invokes the authenticate method, whose execution results in an authentication procedure using standard Java authentication mechanisms.

During the weaving process, the AspectJ compiler identifies all operations that are specified by `authOperations`. In this case, the operations include the method `debit`, which is a method of the `Account` class but does not bear the name of `credit`. The compiler then inserts the body of the advice before the body of this method. This compile-time weaving results in run-time execution of the authentication procedure before the execution of the `debit` method, whose importance requires it to be protected.

The second aspect is a transaction aspect. A transaction assures a set of operations fails or succeeds as a whole, leaving no partial or inconsistent states. The transaction aspect defines a point cut, `transOperation`, which includes the execution of the `debit` method of the `Account` class, as well as the execution of the `transfer` method of the `InterAccountTransferSystem` class. The aspect defines an `around` advice for this point cut. This advice calls a special method `proceed` in its body. The `proceed` method executes the original method as is. If the original method executes successfully, the advice commits the updates. If any exception happens during the execution of the method, the advice rolls back to the initial state that held before any update was made.

The AspectJ compiler weaves the transaction aspect into classes. When the woven application executes, the original `transfer` method of the `InterAccountTransferSystem` is replaced by the `around` advice of the transaction aspect. The advice calls the original method, which executes `debit` method on the outgoing account and `credit` method on the incoming account. If both methods execute successfully, the advice commits the changes. If any method fails, the advice rolls back to the state before any change. It should then notify someone that the transfer is still pending.

3.3 Testing

We apply our approach to test the banking account processing system. In the first step, we develop a set of test cases to test regular classes. The test cases are listed in Figure 3.2. Conforming to the convention of the JUnit framework, each public method whose name begins with `test` is an independent test case. The `setup` method is executed by JUnit to prepare for each test case.

The overall relationships between classes, aspects, and test cases are depicted in Figure 3.3. The dashed lines specify the advice relationship from aspects to methods of classes. The solid arrows describe invocation relationship from test cases to methods of classes.

Our test cases test functionalities of the `Account` class and the `InterAccountTransferSystem` class. For example, the `testDebit` case tests the debit method of the `Account` class, and checks whether the balance after the `debit` method is what it is supposed to be. The `setup` method creates `account2` with initial balance of 300. The test case performs a `debit` operation of 200. `assertEquals` checks whether the resulting balance is 100. Other test cases test the `getAccountNumber` and `credit` method of the `Account` class, and the `transfer` method of the `InterAccountTransferSystem` class when there is sufficient balance and when there is not a sufficient balance.

In the second step, we weave each aspect into the classes and test the aspect. We first test the authentication aspect. Since we have developed test cases in the first step, this step can select and reuse test cases that are relevant to the authentication aspect. Our tool helps us identify relevant test cases. It also checks whether reused test cases that are relevant cover the authentication aspect. That is, whether all methods advised by advice of the authentication aspect are tested by one of these test cases.

We know that the authentication aspect advises the `debit` method of class `Account` and the `transfer` method of class `InterAccountTransferSystem`. Test case `testDebit` invokes the `debit` method, so it is relevant to the authentication aspect. However, itself does not cover the aspect, because the `transfer` method is not executed. Adding the test case `testTransferSufficient` covers the aspect, because both methods are executed by the two test cases. The coverage is depicted in Figure 3.4.

```
public class AccountTest extends
TestCase {
  protected void setUp() {
    account1 = new Account(1, 0);
    account2 = new Account(2, 300);
  }

  public void testAccountNumber()
{.. ..}

  public void testCredit() {.. ..}

  public void testDebit() throws
       InsufficientBalanceException {
     account2.debit(200);
     assertEquals(100,
         account2.getBalance(), 0);
  }

  public void testTransferSufficient()
    throws InsufficientBalanceException
{

InterAccountTransferSystem.transfer
           (account2, account1, 250);
     assertEquals(50,
         account2.getBalance(), 0);
  }

  public void
testTransferInsufficient(){
     .. ..
  }
}
```

Figure 3.2, Tests for Banking Account System

Once test cases are selected, we can execute these test cases. Our tool facilitates automatic execution of selected test cases. Testers can enter login information to check whether authentication is properly applied to desired methods.

We perform unit testing on the transaction aspect in a similar way.

After testing each aspect individually, we move to test the interactions among the aspects in the third step. We weave both the authentication and transaction aspect into classes.

Test cases that are relevant to both aspects are selected to test the interactions between aspects, with the help of our tool. The `debit` method is advised by both aspects, so the test case `testDebit` is selected to touch both aspects.

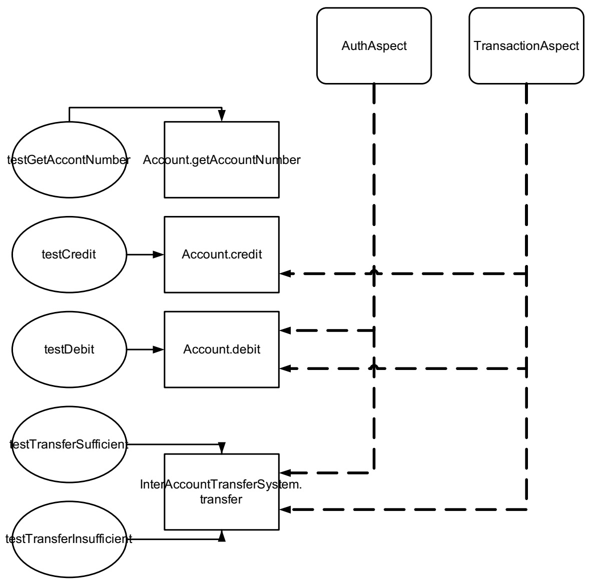

Figure 3.3, Classes, Aspects, and Tests

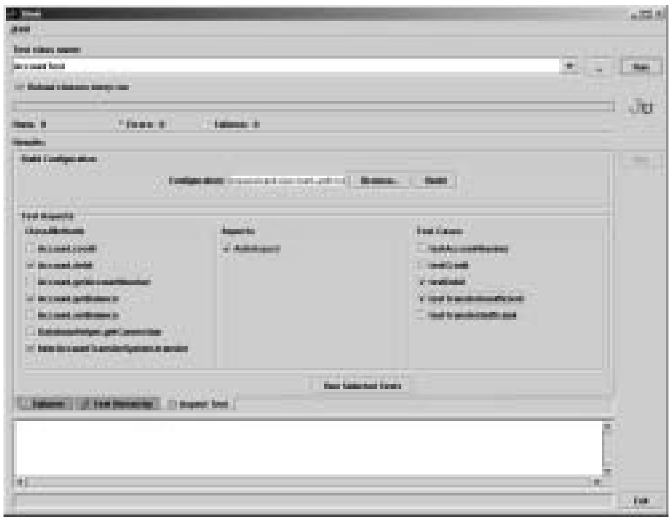

Figure 3.4, Test cases cover authentication aspect

The fourth step of our approach tests the complete system that contains complete aspects and classes. For our sample system that contains only two aspects, the testing activity is the same as that of the third step.

Our case study illustrates our approach and tool can help test practical aspect-oriented software. Relevant test cases are selected to reduce the cost of testing and improve efficiency of testing.

4 Related Work

Call graph construction has been extensively studied. [RY79] is one of the first such algorithms for procedural languages. [GC01] provides an excellent overview for algorithms that construct call graphs for object-oriented programs. [SO03] applies call graph construction to aspect-oriented programs. It uses call nodes to explicitly model invocations on advice by normal methods. The focus of the work is to provide a regular expression-based point cut designator and reduce compile time cost in call stack inspection.

Dependence graphs employ more types of nodes and edges to describe dependency among program constructs. It has been extensively used in procedural and object-oriented programming to perform compilation optimization and program slicing [FOW87, LH96, MMS96, OO84]. Aspect-Oriented System Dependence Graphs [Zh03, ZR03] are a type of dependence graph that describes relationships between classes and aspects. The proposed algorithm constructs a module dependence graph for members of classes and aspects, then uses weaving arcs and point cut vertices to connect those graphs. The work still focuses on program slicing, but it can help analyze useful relationships between aspects and classes for testing. [BM04] suggests undisciplined use of some AspectJ features might lead to imprecise slicing results.

Little research has been conducted to directly address the issue of testing aspect-oriented programs. [CL02] classifies aspects into two categories: observers and assistants. Observers are aspects that would not modify regular classes. Assistants are those aspects that will modify regular classes. Assistants make modular reasoning difficult, and their integration needs more subtle analysis and testing. [KYX03] suggests there are three types of dependency between aspects: orthogonal, unidirectional, and circular. When integrating aspects that are not orthogonal, the order of integration testing is important, and stubs and drivers for missing aspects are required.

[Ka04] considers testing of aspects from a regression testing perspective. They suggest that adding aspects is like evolving software systems, and more careful static analysis and deduction proofs should be applied to avoid the cost of complete retest.

Instead of testing, model checking has been applied to verify aspect-oriented programs. [UT02] utilizes model checkers to verify some subtle problems in a concurrent system, and proposes an AOP-based framework that facilitates such checking. [SK03] proposes a stepwise approach that utilizes aspects to model check programs. Each verification task of model checking is organized into an aspect, which contains both assumptions and results of verification.

5 Conclusion and Future Work

Aspect-Oriented Programming uses aspects to address cross-cutting concerns modularly. Aspects introduce a new dimension of abstraction. The unique relationship between aspects and classes requires new approaches to test aspect-oriented software.

We propose an initial approach towards a comprehensive methodology that can effectively test aspects. The initial approach adapts traditional unit testing, integration testing, and system testing to test aspect-oriented software. We define testing coverage in a way that specifies how well an aspect is tested by a set of test cases. To reduce the cost of testing aspects, we suggest that test cases developed for regular classes can be reused. We propose an algorithm based on control flow analysis to select relevant test cases that test the aspects. When reused test cases cannot cover the aspects under test satisfactorily, new test cases should be developed. We develop a tool to support our approach. The tool can help select relevant test cases and calculate test coverage for the aspects under test. Reusing relevant test cases achieves higher efficiency by reducing testing costs. We use a case study of banking accounts processing to illustrate our approach.

The current approach is only the first step in our exploration. We plan to further develop our approach and algorithm. We are investigating handling more features in unit testing of aspects. First, we are exploring the possibility of treating an advice as a level of abstraction and testing it more explicitly, much like how traditional testing tests intra- and inter- method relationships of a class in object-oriented programs. Second, we are studying what extension is needed to analyze and test the data dependence between classes and aspects introduced by the structural introduction feature [StK03]. Structural introduction is a feature of AspectJ that allows an aspect to introduce new data members for classes. Finally, we are exploring the effect of aspect instantiation policies on testing. Currently there are three choices for aspect instantiation. An aspect can be instantiated just once for the whole system, forming a singleton. An aspect can also be instantiated for each object it relates to. A third option is for an aspect to be instantiated for each control flow. The instantiation policy might affect how test cases should be designed.

6 Acknowledgement

The authors would like to acknowledge and thank the ROSATEA research group in the School of Information and Computer Science at the University of California, Irvine for their valuable insights, in particular the suggestions provided by Tim Standish.

References

[As04] Aspectj, http://www.eclipse.org/aspectj

[Ba98] Ball, T. *On the Limit of Control Flow Analysis for Regression Test Selection.* in Proceedings of Proceedings of ACM SIGSOFT international symposium on Software testing and analysis, p.134-142, 1998.

[BM04] Balzarotti, D. and M. Monga. *Using Program Slicing to Analyze Aspect Oriented Composition.* in Proceedings of 2004 Foundations of Aspect-Oriented Languages Workshop, p.25-30, 2004.

[Ba04] Barat, http://sourceforge.net/projects/barat
[Be90] Beizer, B., *Software Testing Techniques*. 2nd ed. 1990: Van Nostrand Reinhold Co.
[Cl89] Clarke, L.A., et al., *A Formal Evaluation of Data Flow Path Selection Criteria.* Software Engineering, IEEE Transactions on, 1989. **15**(11): p. 1318-1332.
[CL02] Clifton, C. and G.T. Leavens. *Observers and Assistants: A Proposal for Modular Aspect-Oriented Reasoning*. in Proceedings of 2002 Foundations of Aspect-Oriented Languages Workshop, p.33-44, 2002.
[Ec04] Eclipse, http://www.eclipse.org
[FOW87] Ferrante, J., K.J. Ottenstein, and J.D. Warren, *The Program Dependence Graph and Its Use in Optimization.* ACM Trans. Program. Lang. Syst., 1987. **9**(3): p. 319--349.
[GC01] Grove, D. and C. Chambers. *A Framework for Call Graph Construction Algorithms*. in Proceedings of ACM Trans. Program. Lang. Syst., p.685-746, 2001.
[Ju04] Junit, http://www.junit.org
[Ka04] Katz, S. *Diagnosis of Harmful Aspects Using Regression Verification*. in Proceedings of 2004 Foundations of Aspect-Oriented Languages Workshop, p.1-6, 2004.
[Ki97] Kiczales, G., et al. *Aspect-Oriented Programming*. in Proceedings of 11th European Conference on Object-Oriented Programming, p.220-42, 1997.
[KYX03] Kienzle, J., Y. Yu, and J. Xiong. *On Composition and Reuse of Aspects*. in Proceedings of 2003 Foundations of Aspect-Oriented Languages Workshop, p.17-24, 2003.
[La03] Laddad, R., *Aspectj in Action: Practical Aspect-Oriented Programming*. 2003: Manning Publications Co.
[LH96] Larsen, L. and M.J. Harrold. *Slicing Object-Oriented Software*. in Proceedings of Proceedings of the 18th international conference on Software engineering, p.495-505, 1996.
[MMS96] McGregor, J.D., B.A. Malloy, and R.L. Siegmund, *A Comprehensive Program Representation of Object-Oriented Software.* Annals of Software Engineering, 1996. **2**: p. 51-91.
[OO84] Ottenstein, K.J. and L.M. Ottenstein. *The Program Dependence Graph in a Software Development Environment*. in Proceedings of Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments, p.177-184, 1984.
[RW85] Rapps, S. and E.J. Weyuker, *Selecting Software Test Data Using Data Flow Information.* IEEE Transactions on Software Engineering, 1985. **SE-11**(4): p. 367-75.
[RY79] Ryder, B.G., *Constructing the Call Graph of a Program.* IEEE Transactions on Software Engineering, 1979. **SE-5**(3): p. 216-26.
[SO03] Sereni, D. and Oege. *Static Analysis of Aspects*. in Proceedings of Proceedings of the 2nd international conference on Aspect-oriented software development, p.30-39, 2003.
[SK03] Sihman, M. and S. Katz. *Model Checking Applications of Aspects and Superimpositions*. in Proceedings of 2003 Foundations of Aspect-Oriented Languages Workshop, p.51-60, 2003.
[StK03] Storzer, M. and J. Krinke. *Interference Analysis for Aspectj*. in Proceedings of 2003 Foundations of Aspect-Oriented Languages Workshop, p.35-44, 2003.
[UT02] Ubayashi, N. and T. Tamai. *Aspect-Oriented Programming with Model Checking*. in Proceedings of Proceedings of the 1st international conference on Aspect-oriented software development, p.148-154, 2002.
[Zh03] Zhao, J. *Data-Flow-Based Unit Testing of Aspect-Oriented Programs*. in Proceedings of Computer Software and Applications Conference, 2003. COMPSAC 2003. Proceedings. 27th Annual International, p.188-197, 2003.
[ZR03] Zhao, J. and M. Rinard, *System Dependence Graph Construction for Aspect-Oriented Programs*. 2003, Laboratory for Computer Science, MIT.