JAOUT: Automated Generation of Aspect-Oriented Unit Test

Guoqing Xu, Zongyuan Yang, Haitao Huang, Ling Chen and Fengbin Xu
Software Engineering Lab, Department of Computer Science
East China Normal University
{gqxu_02, yzyuan, hthuang}@cs.ecnu.edu.cn

Abstract

Unit testing is a methodology for testing small parts of an application independently of whatever application uses them. It is time consuming and tedious to write unit tests, and it is especially difficult to write unit tests that model the pattern of usage of the application they will be used in. Aspect-Oriented Programming (AOP) addresses the problem of separation of concerns in programs which is well suited to unit test problems. On the other hand, unit tests should be made from different concerns in the application instead of just from functional assertions of correctness or error. In this paper, we firstly present a new concept, application-specific Aspects, which mean top-level aspects picked up from generic low-level aspects in AOP for specific use. It can be viewed as the separation of concerns on applications of generic low-level aspects. Second, this paper describes an Aspect-Oriented Test Description Language (AOTDL) and techniques to build top-level aspects for testing on generic aspects. Third, we generate JUnit unit testing framework and test oracles from AspectJ programs by integrating our tool with AspectJ and JUnit. We use runtime exceptions thrown by testing aspects to decide whether methods work well. Finally, we present a double-phase testing way to filter out meaningless test cases in our framework.

1. Introduction

There is a growing interest in applying program testing to the development process, as reflected by the Extreme Programming (XP) approach [1]. In XP, unit tests are viewed as an integral part of programming. Tests are created before, during, and after the code is written — often emphasized as "code a little, test a little, code a little, and test a little ..." [2]. The philosophy behind this is to use regression tests [3] as a practical means of supporting refactoring.

A unit test suite comprises a set of test cases. A test case consists of a test input and a test oracle, which is used to check the correctness of the test result. Developers usually need to manually generate the test cases based on written or, more often, unwritten requirements. Some commercial tools for Java unit testing, such as ParaSoft's Jtest [4], attempt to fill the gaps not covered by any manually generated unit tests. These tools can automatically generate a large number of unit test inputs to exercise the program. However, no test oracles are produced for these automatically generated test inputs unless developers do some additional work: in particular, they need to write some formal specifications, runtime assertions [5] or more practically, make program invariants generated dynamically with a certain tool like Daikon [6] and use these invariants to improve the test suites generation [7, 8]. However, with the current formal assertions, it is very difficult to generate tests that can model some non-functional features of the program, e.g. the performance of the program and temporal logic of methods' execution.

Aspect-Oriented Programming (AOP) addresses the problem of separation of concerns in programs [9, 10, 11, 12]. Since in AOP, the crosscutting properties are monitored to reflect the program from different aspects, a lot of tasks which have been difficult to be handled in traditional ways are easily done. For example, the performance and methods' execution order problems have been well solved in AOP. Therefore, using a crosscutting property of the program as the criterion to check the correctness of the application in the corresponding aspect is well suited to the unit testing problems.

However, currently in AOP, programmers build generic aspects to monitor certain crosscutting properties for a wide variety of uses including program tracing, rum time assertion checking and etc. This makes it difficult for testers to identify them and make those for testing as test oracles. Therefore, how to build specific testing aspects which can be identified as test oracles becomes the key problem in making the

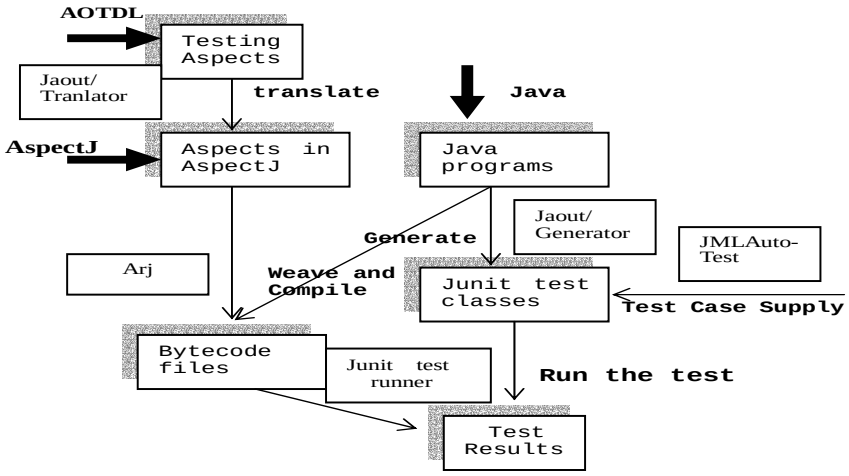

Fig.1. An Overview of the basic technique

aspect-oriented unit testing practical.

We attempt to solve this problem by collecting those aspects in AOP for the same use into application-related top-level aspects. We call these aspects the application-specific aspects. For example, aspects for tracing can be collected and made into Tracing Aspects, aspects for pre and post condition assertions are to be built as Assertion Aspects and etc. Building application-specific aspects can be viewed as the separation of concerns on applications of generic aspects. A kind of application-specific aspects share some common features. In this way, we can build Testing Aspects which will send runtime messages which can be received by unit test programs and identified as test oracles.

Fig.1 illustrates the basic technique used in our approach to generating the unit test. The testing aspects described by Aspect-Oriented Test Description Language (AOTDL) can be translated by our tool JAOUT/translator as low-level aspects in AspectJ. Then after weaved with Java programs, the aspects are compiled to bytecode files (class files). The tool JAOUT/generator generates the corresponding JUnit test classes from the program to be tested. These test codes can serve as test oracles. Finally, fed with the test inputs generated automatically by JMLAutoTest [14], the unit test is run and the results are judged by the runtime exceptions thrown from the testing aspects we made.

This paper makes the following three major contributions:

- We describe an Aspect-Oriented Test Description Language (AOTDL) to help build Testing Aspects in AOP.
- Our tool JAOUT makes the translation from testing aspects to generic aspects in AOP and inserts them into the AspectJ programs. Then JAOUT generates the test classes for Junit from AspectJ programs and the generated test codes can serve as test oracles.
- We use a novel approach, double-phase testing, to filtering out the meaningless test cases.

The rest of this paper is organized as follows: section 2 briefly introduces the basic concepts in AspectJ and section 3 describes how to build testing aspects with AOTDL. Then in section 4, we describe the JUnit framework. We present the generation of test oracle and test cases in detail in section5 and the techniques of double-phase testing in section6. After section 7 describes the related work, we conclude our approach in section 8.

2. AspectJ

In this paper, we use AspectJ as our target language to show the basic idea of aspect-oriented unit testing. AspectJ [13] is a seamless aspect-oriented extension to Java. AspectJ adds some new concepts and associated constructs are called join points, pointcuts, advice, and aspect.

The *join point* is an essential element in the design of any aspect-oriented programming language since join points are the common frame of reference that defines the structure of crosscutting concerns. The *join points* in AspectJ are well-defined points in the execution of a program. A *pointcut* is a set of joint

```
Class Stack{
  public void init(){...}
  public void push(Node n){...}
  ...
}

Aspect TempLogic{
  protected boolean isInitialized = false;
   //method push is called
  pointcut pushReached(Stack st):
      target(st)&&call(void
Stack.push(Node));
  //method init is called
  pointcut initReached(Stack st):
      target(st)&&call(void
Stack.init(void));
  //advice after init is called
  after(Stack st):initReached(st){
    isInitialized = true;
 }
  //advice before push is called
  before(Stack st)
       throws NotInitializedException:
  pushReached(st){
    if(!isInitialized)
     throw new NotInitializedException();
 }
}
```

Fig. 2. An Aspect providing advices for the temporal logic of methods execution in *Stack*

points that optionally exposes some of the values in the execution of these joint points. AspectJ defines several primitive *pointcut* designators and pointcuts can be defined according to these combinations.

Advice is a method-like mechanism used to define certain codes that is executed when a point cut is reached. There are three types of advice, that is, *before*, *after* and *around*. In addition, there are two special cases of after advice, *after returning* and *after throwing*, corresponding to the two ways a sub-computation can return through a join point.

Aspects are modular units of crosscutting implementation. *Aspects* are defined by aspect declarations, which have a similar form of class declarations. Aspect declarations, as well as other declarations such as method declarations, that are permitted in class declarations.

An AspectJ program is composed of two parts: (1) non-aspect code which includes some classes, interfaces, and other language constructs as in Java, (2) aspect code which includes aspects for modeling crosscutting concerns in the program. Moreover, any implementation of AspectJ is to ensure that aspect and non-aspect code run together in a property coordination fashion. Such a progress is called aspect weaving and involves making sure that applicable advice runs at the appropriate join points.

Fig.2 illustrates a sample aspect which provides advices for the methods execution order in the class *Stack*. The advices defined in this aspect require that the *push* method can not be executed if the *init* method is not called.

3. Building Testing Aspects

In this section, we present how to build the application-specific aspects for testing with AOTDL.

3.1 AOTDL

AOTDL explicitly specifies the advices for the criteria of meaningless test cases and test errors. Fig.3 illustrates the AOTDL representation of the TempLogic aspect which we have mentioned above. The major differences between two representations in Fig.2 and 3 lie with the fact that the aspect in Fig.2 is a low-level AspectJ aspect which can be of any uses, e.g. tracing, asserting, logging and whatever. But the aspect in Fig.3 is the application-specific aspect which is specifically built for testing. The *Meaningless* and *Error* units contain the advices for criteria of meaningless[1] test cases and test errors respectively. We use the syntax as follows:

advicetype(arguments): *pointcuts*: *conditions*: *message*

conditions is a boolean expression which means the conditions in which the current test case is considered as meaningless or the current test fails. The *message* means the printed message when the conditions happen. To enhance the expressiveness of *conditions* clause, AOTDL supports most kinds of boolean expressions used in formal languages predicts [15] including *forall*, *exist*, function calls returning boolean values and etc. And we are still working on supporting more expressions to make the language convenient and expressive enough for testers. The declarations of pointcuts, fields and all the other advices which do not directly affect the criteria of what is a meaningless case and what is a failed test are put into the *Utility* unit.

We use the ANTLR [41] parser generator to generate the translator in our tool JAOUT/translator. Although the syntax of AOTDL seems simple, it bridges the gap between abstract application-specific

[1] Meaningless test cases means test inputs that are not in the range required by the current test, e.g. a negative number for a method which returns the square root of the received argument.

```
TestingAspect TempLogic{
 // all pointcuts and other utility advices are
declared
 // in the Utility unit
  Utility{
   protected boolean isInitialized = false;
  //push is reached
   pointcut pushReached(Stack st):
     target(st)&&call(void
Stack.push(Integer));
  //init is reached
   pointcut initReached(Stack st):
     target(st)&&call(void Stack.init(void));
   after(Stack st):initReached(st){
     isInitialized = true;
   }
  }
  MeaninglessCase Advice{
   //advices for specifying criteria of
   //meaningless test cases
    before(Stack s):
        pushReached(s):
            s.getSize()>=MAX:"Overflow";
    ...
 }
  Error Advice{
  //advices for specifying criteria of test
  //errors
   before(Stack s):
        pushReached(s):
          !isInitialized:"Not Initialized";
    ...
 }
}
```

Fig.3. The AOTDL representation of the TempLogic Aspect

Aspects and detailed language level aspects of the program, hence can be viewed to help the separation of concerns on applications of generic aspects in AOP.This idea can also be used in making other application-related tools to extract top-level application aspects from the generic aspects.

3.2 Translation of testing aspects

JAOUT/translator is made to translate the testing aspects represented by AOTDL into generic aspects in AspectJ. The translation of TempLogic testing aspect shown in Fig.3 is illustrated in Fig.4. Definitions in *Utility* unit are not changed since they do not affect test oracles. The advices defined in *meaningless* and *error advice* unit throw a *MeaninglessTestCaseException* and *TestErrorException* respectively when the specified conditions are reached.

We make clear that AOTDL is the extension of AspectJ in making application-specific aspects for testing. Our translator also supports the mixture of the testing aspect specified by AOTDL and generic aspects

```
public Aspect TempLogic{
  // Definitions in Utility unit are not changed
   protected boolean isInitialized = false;
   //push is reached
   pointcut pushReached(Stack st):
     target(st)&&call(void Stack.push(Integer));
    //init is reached
    pointcut initReached(Stack st):
      target(st)&&call(void Stack.init(void));
    after(Stack st):initReached(st){
    isInitialized = true;
   }
  //meaningless advices
  before(Stack s)
     throws MeaninglessTestInputException:

  pushReached(s){
   if(s.getSize()>=MAX){
    MeaninglessTestInputException ex= new
     MeaninglessTestInputException("overflow");
       ex.setSource("TempLogic");
    }
  }
  //error advices
  before(Stack s)throws TestErrorException:
                  pushReached(s){
    if(!isInitialized){
       TestErrorException ex =new
          TestErrorException("Not Initialized");
       ex.setSource("TempLogic");
  }
 }
}
```

Fig.4. The Translation of TempLogic Testing Aspect

in AspectJ. After translated by JAOUT/translator and then weaved with the *Stack* program and compiled by arj, the byte codes (.class) files are generated.

4. Assumptions about testing framework

Our approach assumes that unit tests are to be run for each method of each class being tested. We assume that the framework provides test methods, which are methods used to test the methods of the class under test. For convenience, we will assume that test methods can be grouped into test classes. In our approach, each test method executes several test cases for the method it is testing. Thus we assume that a test method can indicate to the framework whether each test case fails, succeeds, or was meaningless. The outcome will be meaningless if a *MeaninglessTestInputException* occurs for the test case. We also assume that there is a way to provide test data to test methods.

Following JUnit's terminology, we call this a test fixture. A test fixture is a context for executing a test; it typically contains several declarations for test inputs

```
import junit.framework.*;

public class PersonTest extends TestCase
{
  private Stack s;
  public PersonTest(String name){
    super(name);
}

public void testPush() {
  int oldsize = s.size();
  s.push(10);
  assertEquals(s.size(), oldsize+1);
}

protected void setUp() {
  s= new Stack();
}
protected void tearDown() {

}
public static Test suite() {
  return new TestSuite(StackTest.class);
}
public static void main(String args[]) {
  String[] testCaseName =
    {PersonTest.class.getName()};
     junit.textui.
        TestRunner.main(testCaseName);
}
}
```

Fig.5. A sample JUnit class

and expected outputs. For the convenience of the users of our approach, we assume that it is possible to define a global test fixture, i.e., one that is shared by all test methods in a test class. With a global test fixture, one needs ways to initialize the test inputs, and to undo any side effects of a test after running the test.

JUnit is a simple, useful testing framework for Java [39]. In JUnit, a test class consists of a set of test methods. The simplest way to tell the framework about the test methods is to name them all with names beginning with "test".

The framework uses introspection to find all these methods, and can run them when requested. Fig. 5 is a sample JUnit test class, which is designed to test the class Stack. Every JUnit test class must be a subclass, directly or indirectly, of the framework class TestCase. The class TestCase provides a basic facility to write test classes, e.g., defining test data, asserting test success or failure, and composing test methods into a test suite.

One uses methods like assertEquals, defined in the framework, to write test methods, as in the test method testPush. Such methods indicate test success or failure to the framework. For example, when the arguments to assertEquals are not equal, the test fails. Another such framework method is fail, which directly indicates test failure. JUnit assumes that a test succeeds unless the test method throws an exception or indicates test failure. Thus the only way a test method can indicate success is to return normally.

JUnit provides two methods to manipulate the test fixture: setUp creates objects and does any other tasks needed to run a test, and tearDown undoes otherwise permanent side-effects of tests. For example, the setUp method in Fig.5 creates a new Stack object, and assigns it to the test fixture variable s. The tearDown method can be omitted if it does nothing. JUnit automatically invokes the setUp and tearDown methods before and after each test method is executed (respectively).

The static method suite creates a test suite, i.e., a collection of test methods. To run tests, JUnit first obtains a test suite by invoking the method suite, and then runs each test method in the suite. A test suite can contain several test methods, and it can contain other test suites, recursively. Fig. 5 uses Java's reflection facility to create a test suite consisting of all the test methods of class StackTest.

5. Test Oracles and Test Cases Generation

This section presents the details of our approach to automatically generating a JUnit test class from a Java class weaved with aspects. We firstly describe how test outcomes are determined. Then we describe the protocol and techniques for the user to supply test data to generated test oracles. Finally, we discuss the automatic generation of test methods and test classes.

5.1 Determining test outcomes

The outcome of a call to M for a given test case is determined by whether translated testing aspects throw exceptions during M's execution, and what kind of exception is thrown. If no exception is thrown, then the test case succeeds (assuming the call returns), because there was no meaningless case/error found, and hence the call must have satisfied specifications defined in testing aspects.

Similarly, if the call to M for a given test case throws an exception that is not an *MeaninglessTestInput* or *TestError* exception, then this also indicates that the call to M succeeded for this test case since they are thrown by other aspects or the method itself, but not testing aspects. Hence if the call to M throws such an exception instead of a meaningless or error exception, then the call must have satisfied specifications of testing aspects. With JUnit, such exceptions must, however, be caught by the test method testM, since any such exceptions are interpreted by the framework as signaling test failure. Hence, the testM method must catch and ignore all exceptions that are not meaningless and error exceptions.

If the call to M for a test case throws a *TestError* exception, then the method M is considered to fail that test case. If a *MeaninglessTestInput* exception is caught, the current test case is considered as a meaningless test case and therefore, is ignored.

5.2 Supplying test cases

Given a Java class *M.java*, JAOUT/Generator generates three classes: *M_Aspect_Test*, *M_Aspect_TestCase* and *M_Aspect_TestClient*. *M_Aspect_Test* is a JUnit test class to test all methods in class M. *M_Aspect_TestCase* is a test case provider, which extends *M_Aspect_Test* and initialize test fixture. *M_Aspect_TestClient* is the test client in which test cases can be generated automatically by JMLAutoTest [14] tools.

Similar to JMLUnit test framework [5], the test fixture for the class C is defined as:

C[ ] receivers; T_1[ ] vT_1; ... ; T_n[ ] vT_n;

The first array named *receivers* is for the set of receiver objects (i.e., objects to be tested) and the rest are for argument objects. If a particular method has formal parameter types A1;A2; ...;Am, where each Ai is drawn from the set {T1,..., Tn}, then its test cases are:

{<receivers[i],vA1[j1],...,vAm[jm]>
| 0≤i< receivers.length, 0≤j1 < vA1.length,..., 0≤jm < vAm.length }

The receiver and argument objects are initialized by the method *init_receivers* and *init_vTi* in the test case provider class with the test cases obtained from the test client as shown in Fig.6. Testers can write the codes in the *makeTestCases_CaseType* method defined in the test client class with the help of utility classes provided by JMLAutoTest to let test cases generated automatically. According as conditions specified by testers, JMLAutoTest generate all objects including those with linked data structures.

5.3 Test method

There will be a separate test method, testM for each target instance method (non-static), M, to be tested. The purpose of testM is to determine the outcome of calling M with each test case and to give an informative message if the test execution fails for that test case. The method testM accomplishes this by invoking M with each test case and indicating test failure when the testing aspects throw a *TestError*

```
//The following codes appear in the test case
//provider  ((Stack_Aspect_TestCase.java)  for
//testing the class Stack.

public class Stack_Aspect_TestCase
       extends Stack_Aspect_Test{

//an object of test client
  protected AspectTestClient  myClient
         = new AspectTestClient();
// test cases container
  protected Hashtable param
          = new Hashtable();

public static void main(String[] args){
 //Initialize hashtable param in order to
 //receive test cases
 // Get test cases from TestClient
  param.put("Integer",myClient.
             makeTestCases_Integer());
  ...

}
 //Initialize receivers
 public void init_Receivers(){
   receiver = new Stack[4];
   receiver[0]= ...;
   receiver[1]= ...;
   ...
}
 //Initialize test inputs of type Integer
public void init_vInteger (){
   // get test cases from the hashtable
   vT1 = (Integer[])param.get("Integer");
   ...
 }
}
```

Fig.6. Generated Codes skeleton in Test Case Provider (Stack_Aspect_TestCase) for supplying the test cases of type Integer

exception. Test methods also note if test cases were rejected as meaningless when a *MeaninglessTestInput* exception was caught.

What Fig.7 illustrates is the test method for the method M. The differences between this method and the test method in Jmlunit [5] framework are that the *MeaninglessTestInputException* and *TestErrorInputException* are replaced for the *PreconditionEntryError* and *PostconditionError* and the exceptions used in our testing framework are thrown by **Testing Aspects** instead of a runtime assertion checker. Since patterns difficult to be modeled with traditional formal specifications are well treated as recognized as crosscutting properties in aspects, our approach may be adapted in more situations than a testing framework based on formal specifications exemplified by Jmlunit. Fig.8. illustrates the test result when the *push* method is tested with the stack not initialized.

```
public void testM() {
  final A1[] a1 = vA1;
    : : :
  final An[] an = vAn;
  for (int i0 = 0; i0 < receivers.length; i0++)
   for (int i1 = 0; i1 < a1.length; i1++)
      : : :
      for (int in = 0; in < an.length; in++) {
       if (receivers[i0] != null) {
        try {
           receivers[i0].M(a1[i1], ..., an[in]);
        }
        catch (MeaninglessTestInputException e) {
  /* ... tell framework the test case was meaningless ... */
         result.meaninglesscases++;
         continue;
        }
      /*  ... tell framework the current test fails*/
        catch (TestErrorException e) {
            String msg =  "In testing aspect  "+
                     e.getSource()+":"+e.getMessage();
                     system.err.println(msg);
            result.errors++;
        }
        catch (java.lang.Throwable e) {
            continue; // success for this test case
        }
        finally {
           setUp(); // restore test cases
        }
     } else {
   /* ... tell framework test case was meaningless, since the
        test cases are not initialized ... */
    }
    : : :
   }
  }
}
```

Fig.7. The test method for the corresponding method M

6. Double-phase Testing

There are a number of tools that can automate the test case generation. However, generated test case space may often too large to be totally exercised to the program, because there might exist a large number of meaningless test cases for the current test. Existing test frameworks do not take into account this problem. In these frameworks, testers usually generate test cases with a tool, feed the program with the whole test input space and let the framework decide whether the current test input is meaningful. However, handling too many meaningless test cases is a waste of time and greatly affects the quality of the test outcomes. The extra time wasted is subject to the way that the test framework deals with the meaningless test cases. For

```
...in push
false
F
Time: 0.06
There was 1 failure:
1)
testPush(sample.Stack_Aspect_TestCase)junit.framework.Assertion
FailedError: In Testing Aspect TempLogic: Not Initilized!
  at ample.Stack_Aspect_Test.testPush
                        (Stack_Aspect_Test.java:155)
  at sun.reflect.NativeMethodAccessorImpl.invoke0
                        (Native Method)
  at sun.reflect.NativeMethodAccessorImpl.invoke
                        (NativeMethodAccessorImpl.java:39)
  at sun.reflect.DelegatingMethodAccessorImpl.invoke
                        (DelegatingMethodAccessorImpl.java:25)
  at sample.Stack_Aspect_Test.run(Stack_Aspect_Test.java:26)
  at sample.Stack_Aspect_TestCase.main
                         (Stack_Aspect_TestCase.java:24)

FAILURES!!!
Tests run: 3,  Failures: 1,  Errors: 0, Meaningless:0
```

Fig.8. The Test Result when the *push* is run without initialized

example, for the JMLUnit [5] test framework , this part is the time to execute the precondition checking methods since in JMLUnit test cases which violate pre-condition of the method to be tested are considered as meaningless test cases. In our JAOUT framework, this is the time to execute the meaningless advices in Testing Aspects at the specified joint points.

Our technique to filter out meaningless test cases includes three parts: making Operational Profile, pre-test and final test. The major idea of double-phase testing is to use two phases of testing based on statistic. First, testers make a criterion called Operational Profile to divide the whole generated test case space into several partitions. During the first phase, a relatively small number of test cases taken from each partition make several groups. Test suite should be run for several times and each group of test cases is supplied to test respectively to show the number of meaningless cases contained in each partition. During the second phase, a large number of test cases taken from each partition according to the proportion of meaningless test cases obtained in the first phase are mixed together to be supplied to the test.

Double-phase testing uses the first phase pre-test as the cost for saving the time and increasing the test quality in the second phase final test. Double-phase testing is especially suited for the black box test and the test with a large test case space since the bigger the test case space, the more time will be wasted in handling meaningless test cases.

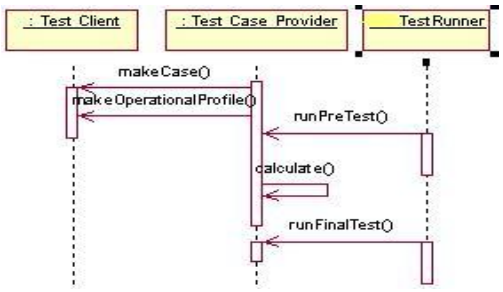

Fig. 9. The working sequence of the Double-phase testing

Fig.9 illustrates the working sequence of the double-phase testing used in our JAOUT framework. Testers use APIs provided by a tool (JMLAutoTest[14] in our framework) to generate test case space and make Operational Profile to partition the space. Test case provider uses the portioned test cases to initialize JUnit test case arrays. After the pre-test is run, Test runner calculates the distribution of meaningless test cases in each partition, and then reorganizes the test cases to run the final test.

Experimental Results. We undertook an experiment to illustrate the performance of double-phase testing. The method to be tested is *BinaryTree findSubTree(BinaryTree t, Node n).* This method is to find a sub-tree whose root is represented by node n, in a parent tree t. We used JMLAutoTest to generate all binary trees with the number of nodes from five to eight for the argument t. We also generated twelve nodes whose IDs are from 0 to 11 for the argument n. We made the Test Aspects which specified that if the argument node n cannot be found in the binary tree t, the current test inputs are meaningless. It is clear that if a node n whose ID is larger than eight is fed to the method, the current test case will surely be meaningless. Table 1 shows the performance comparison between double-phase testing used in our framework and conventional testing ways. There are two points we can see very clearly. The first point is in double-phase testing, there are almost no meaningless test cases existing in the final test while using conventional ways can not avoid meaningless cases (Column 2 and 5). The second point is that the extra time has been saved in the double-phase way (Column 4 and 7). When test case space is not very large, this point cannot be seen. Actually, the time used in double-phase way is more than that in traditional testing when there are five nodes in the binary tree. However, with the growing of the number of test inputs, this advantage becomes clear, since the time cost in handling meaningless test cases greatly exceeds the time used in pre-test. From this experiment, we can find out double-phase testing is not only a theoretical proposal, but also can be practical in the unit testing.

Table 1. Performance comparison between the double-phase testing and the conventional testing ways.

	Double-phase way			Conventional way		
nodes in binary tree	**meaning-gful /total in final test**	**time in the first phase (s)**	**total time of the test (s)**	**meaning-ful /total in final test**	**time in the first phase (s)[2]**	**total time of the test (s)**
5	410/492	0.079	0.266	420/1008	0	0.219
6	1572/1572	0.188	0.422	1584/3168	0	0.468
7	5136/5136	0.36	0.766	6006/10296	0	1.25
8	17148/17148	0.703	2.016	22880/34320	0	3.484

7. Related Work

7.1 Aspect-Oriented programming

Aspect-oriented software development (AOSD) [16] is a new technique to support separation of concerns in software development [1, 2, 3, 4]. This programming methodology is created by Xerox Palo Alto Research Center, and firstly presented in ECOOP97 [10]. A lot of researchers have conducted researches on AOP. One of the major areas include the attempt to improve the programming methodology and language itself, such as modular reasoning in AOP [17, 18, 19], model checking AOP systems [20, 21, 22] and exploiting type systems to enable reasoning about aspect-oriented programs [23]. The rest of them mainly emphasize the application of AOP, ranging from software architecture [24] to large-scale software security [25]. And currently the languages which support AOP mainly include AspectJ [13] for Java, Aspect C++ [26] for C++, AspectS [27] for Squeak/Smalltalk and Aspects [28] for Python.

7.2 Specification-based Testing

There is a large body of research on specification-based testing. An early paper by Goodenough and Gerhart [29] emphasizes its importance. Many projects automate test generation from specifications, such as Z specifications [30, 31], UML statecharts [32, 33], or

[2] All the time in this column is zero because there is no the first phase test in conventional testing.

ADL specifications [34, 35]. These tools don't generate test cases.

The TestEra framework [36] generates Java test cases from Alloy [37] specifications of linked data structures. TestEra uses the Alloy Analyzer (AA) [38] to automatically generate method inputs and check correctness of outputs, but it requires programmers to learn a specification language much different than Java.

Boyapati, Khurshid and Marinov describe Korat [40] which perform automated generation of test cases for Java programs with formal specifications. Korat can generate linked data structures based on additional Java predicates. Cheon and Leavens [5] describe automatic translation of JML specifications into test oracles for JUnit [39]. This framework automates execution and checking of methods. However, the burden of test case generation is still on programmers: they have to provide sets of possibilities for all method parameters. JMLAutoTest [14] solves this problem by generating test cases based on JML invariant clause specified in the class, and therefore does not require testers to write a special predicate method, such as repOK() in Korat.

However, all these approaches are based on traditional formal specifications, some patterns such as performance and temporal logic of methods executions are very hard to be modeled and therefore, these testing frameworks can not be used to test these specific aspects of programs.

8. Conclusions

Automatically generating unit tests is an important approach to making unit test practical. However, existing specification-based tests generation methods can only test the program behavior specified by invariants. Since the program invariants only focus on functional behaviors, patterns related to non-functional properties of the program can not be modeled, and therefore, existing specification-based test generation can not test these special aspects of the program.

In the paper, we take a different perspective and present an approach to generating the unit testing framework and test oracles from aspects in AOP. First, we describe a new concept, application-specific aspect, which means the top-level aspect picked up from generic aspects in AOP. This can be viewed as the separation of concerns on specific application of common AOP's aspects. Then we discuss an Aspect-Oriented Test Description Language (AOTDL) to build the application-specific aspects for testing, namely testing aspects. AOTDL explicitly specifies the properties for testing which can be translated into the common aspects in AspectJ. After weaving and compiling programs, we automatically generate the unit testing codes which can serve as test oracles. Test outcomes are decided on different exceptions thrown by testing aspects. Finally, we present a novel testing approach, double-phase testing to avoid the meaningless test cases before the final test is run. From the experimental results, we make it clear that double-phase testing filters out the meaningless test inputs and saves the time.

Acknowledgement

We thank Gary T. Leavens for his assistance in our use of Jmlunit framework and his enlightenment in the idea of aspect-oriented unit testing. We are also grateful to all members in Software Engineering Lab for their helpful discussion and comments on this paper.

Reference

[1] Beck K., Extreme Programming Explained. Addison-Wesley, 2000.

[2] Beck K. and Gamma E., Test infected: Programmers love writing tests. *Java Report, 3(7),* July 1998.

[3] Korel B. and Al-Yami A. M., Automated regression test generation. *In Proc of ISSTA 98,* Clearwater Beach, FL, pages 143–152. IEEE Computer Society, 1998.

[4] Parasoft Corporation, Jtest manuals version 4.5, http://www.parasoft.com/, October 23, 2002.

[5] Cheon Y. and Leavens G. T., A simple and practical approach to unit testing: The JML and JUnit way, *In Proc of 16th European Conference Object-Oriented Programming (ECOOP02), pp. 231-255*., 2002.

[6] Ernst, M. D., Cockrell, J., Griswold, W. G., Notkin, D. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering, vol. 27, no. 2,* Feb. (2001) 1-25

[7] Xie T. and Notkin D., Tool-Assisted Unit Test Selection Based on Operational Violations, *In Proc of 18th IEEE International Conference on Automated Software Engineering (ASE'03),* Montreal, Canada, Oct. 2003.

[8] Xie T. and Notkin D., Mutually Enhancing Test Generation and Specification Inference. *In Proc of 3th International Workshop on Formal Approaches to Testing of Software (FATES'03),* Montreal, Canada, Oct. 2003.

[9] Bergmans L. and Aksit M, composing crosscutting concerns using composition filters, *Communications of the ACM, Vol.44, No.10, pp.51-57,* Oct.2001.

[10] Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C., Loingtier J.-M., and Irwin J.. Aspect-oriented programming. *In Proc. ECOOP'97, LNCS vol. 1241,* Springer-Verlag, 1997.

[11] Lieberherr K.J., Orleans D., and Ovlinger J., Aspect-Oriented Programming with Adaptive Methods, *Communication of the ACM, Vol.44, No.10, pp.39-41,* Oct.2001.

[12] Ossher H. and Tarr P., Multi-Dimensional Separation of Concerns and the Hyperspace Approach, *In Proc. the*

Symposium on Software Architectures and Component Technology: The State of the Art in Software Development, Kluwer, 2001.
[13] The AspectJ team, The AspectJ Programming Guide, http://eclipse.org/aspectj, May 2004.
[14] Xu G. and Yang Z., JMLAutoTest: A Novel Automated Testing Framework Based on JML and JUnit., *In Proc. 3rd International Workshop on Formal Approaches to Testing of Software(FATES'03), pp. 118-127*, Montreal, Canada, Oct.2003., also in *LNCS vol.2931*, Springer-Verlag, Jan. 2004.
[15] Leavens G. T., Baker A. L., and Ruby C.. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report TR 98-06i, Department of Computer Science, Iowa State University, June 1998. (last revision: Aug 2001).
[16] AOSD website, http://aosd.net, May 2004.
[17] Blair, L., Monga, M. Reasoning on AspectJ Programmes, *GI-AOSDG 2003*, Essen, Germany.
[18] Clifton, C., and Leavens, G. T., Observers and Assistants: A Proposal for Modular Aspect-Oriented Reasoning, Technical Report TR#02-04, Department of Computer Science, Iowa State University, March 2002.
[19] Clifton, C., and Leavens, G. T., Obliviousness, Modular Reasoning, and the Behavioral Subtyping Analogy, Technical Report TR#03-01A, Department of Computer Science, Iowa State University, March 2002.
[20] Ubayashi, N., Tamai, T., Aspect-oriented programming with model checking, *In Proc. the 1st international conference on Aspect-oriented software development*, April 22-26, 2002, Enschede, The Netherlands
[21] Dingel, J., Garlan, D., Jha, S., Notkin, D., Reasoning about implicit invocation, *In Proc. the 6th ACM SIGSOFT international symposium on Foundations of software engineering*, November 1998
[22] Bradbury, J. S., Dingel, J., Evaluating and Improving the Automatic Analysis of Implicit Invocation Systems, *In Proc. of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE 2003)*, Helsinki, Finland, Sept. 2003.
[23] Aldrich, J., A Typed, Modular Foundation for Aspect-Oriented Programming, I*n Proc. Foundations of Aspect-Oriented Languages Workshop (FOAL'04), pp.7-18*, Apr.2004.
[24] Aldrich J., Chambers G., and Notkin D., ArchJava: Connecting Software Architecture to Implementation, *In Proc. International Conference on Software Engineering (ICSE'02)*, Orlando, FL, 2002.
[25] Viega J., Bloch J.T. and Chandra P., Applying Aspect-Oriented Programming to Security, *Cutter Journal, Vol. 14, No.2*, Feb. 2001
[26] Aspect C++ team, http://www.aspectc.org/, May 2004.
[27] AspectS team, http://www.prakinf.tu-ilmenau.de/~hirsch/Projects/Squeak/AspectS/, May 2004.
[28] Aspects team, http://www.logilab.org/projects/aspects/, May 2004.
[29] Goodenough J. and Gerhart S., Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, June 1975.
[30] Horcher H.-M., Improving software tests using Z specifications. *In Proc. 9th International Conference of Z Users*, The Z Formal Specification Notation, 1995.
[31] Spivey J.M., The Z Notation: A Reference Manual. Prentice Hall, second edition, 1992.
[32] Offutt J. and Abdurazik A., Generating tests from UML specifications. *In Proc. Second International Conference on the Unified Modeling Language*, Oct. 1999.
[33] Rumbaugh J., Jacobson I., and Booch G., The Unified Modeling Language Reference Manual. Addison-Wesley Object Technology Series, 1998.
[34] Chang J. and Richardson D. J., Structural specification-based testing: Automated support and experimental evaluation. *In Proc. 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, pages 285–302, Sept. 1999.
[35] Sankar S. and Hayes R., Specifying and testing software components using ADL. Technical Report SMLI TR-94-23, Sun Microsystems Laboratories, Inc., Mountain View, CA, Apr. 1994.
[36] Marinov D. and Khurshid S., TestEra: A novel framework for automated testing of Java programs. *In Proc. 16th IEEE International Conference on Automated Software Engineering (ASE)*, San Diego, CA, Nov. 2001.
[37] Jackson D., Shlyakhter I., and Sridharan M.. A micromodularity mechanism. *In Proc. 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, Vienna, Austria, Sept. 2001.
[38] Jackson D., Schechter I., and Shlyakhter I., ALCOA: The Alloy constraint analyzer. *In Proc. 22nd International Conference on Software Engineering (ICSE)*, Limerick, Ireland, June 2000.
[39] JUnit website, http://www.junit.org, May 2004.
[40] Boyapati C., Khurshid S. and Marinov D., Korat: Automated Testing Based on Java Predicates. In *Proc. ACM International Symposium on Software Testing and Analysis (ISSTA 2002)*, pages 123-133., July 2002.
[41] ANTLR website, http://www.antlr.org, May 2004.