JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2009**

Vol. 8, No. 3, May-June 2009

Automated State-Based Unit Testing for Aspect-Oriented Programs: A Supporting Framework

Mourad Badri, Linda Badri and Maxime Bourque-Fortin,
Department of Mathematics and Computer Science, University of Quebec at Trois-Rivières, Quebec, Canada.

Abstract

Interactions between aspects and classes are a new source for faults. Existing object-oriented testing techniques are not adequate for testing aspect-oriented programs. As a consequence, new testing techniques must be developed. We present, in this paper, a state-based unit testing technique for aspect-oriented programs and associated tool (AJUnit). The technique focuses on the integration of one or several aspects to a class. The objective is to ensure that the integration is done without affecting the original behavior of the class. AJUnit, based on the model of JUnit, generates testing sequences covering an aspect(s)-class block of code. It also supports the execution and verification of the generated sequences. We focus on AspectJ programs. Testing an aspect(s)-class block is done incrementally. Furthermore, the generated sequences are archived. In the case of a change instantiated on a class or on one of its related aspects, only the testing sequences corresponding to the affected parts of the code are retested. The same approach is followed when introducing a new aspect influencing the class under test. The technique is based on several testing criteria that we defined. The generation and verification process of the testing sequences is completely automated.

1 INTRODUCTION

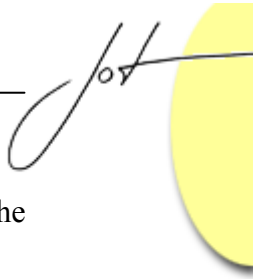
Existing object-oriented programming languages suffer from a serious limitation in modularizing adequately crosscutting concerns in a program. Many concerns crosscut several classes in an object-oriented program. The corresponding code is often dispersed and duplicated in many classes. This affects the modularity of the code and makes programs difficult to understand, to test, to maintain and to reuse [Balt 01]. Aspect-oriented programming provides new mechanisms that explicitly capture crosscutting concerns into modular units called *aspects* [Elra 01, Alex 04, Ajws 05]. This tends to improve programs modularity [Walk 99, Kicz 01, Ajpg 02, Zaka 02, Mort 04, Alex 04, Zhao 04] achieving its usual benefits. AspectJ, the most used aspect-oriented language, is

Mourad Badri, Linda Badri, Maxime Bourque-Fortin: "Automated State-Based Unit Testing for Aspect-Oriented Programs: A Supporting Framework", in *Journal of Object Technology*, vol. 8, no. 3, May-June 2009, pp. 121-126 http://www.jot.fm/issues/issue_2009_05/article4/

an extension of Java. In spite of the many claimed benefits that the aspect paradigm seams offering, it remains that it is not yet mature. Aspect paradigm introduces, in fact, new dimensions in terms of control to software engineering. Interactions between aspects and classes are, in fact, a new source for faults [Alex 04, Zhou 04, Mort 04]. Aspects can have indirect effects that may be difficult to detect. These effects can also lead, in certain cases, to conflicts between aspects. Existing object-oriented testing techniques are not adequate for testing aspect-oriented programs [Alex 02, Zhao 04, Mass 07]. Thus, new testing techniques must be developed. Testing is a crucial issue in software development. It is an essential task to ensure software quality [Beiz 90]. Aspects are weaved in the control flow of a program. Their behavior often depends on the woven context. An aspect can not be tested alone [Alex 04, Mass 07]. Moreover, the relationship between an aspect and related classes is fundamentally different from the one existing between classes in an object-oriented program [Alex 04, Mort 04]. In an aspect-oriented program, something different occurs since integration rules are defined in aspects.

We present, in this paper, a state-based unit testing technique for aspect-oriented programs and associated tool (called AJUnit). The technique is based on the dynamic behavior of classes and related aspects. It supports both the generation and verification of testing sequences. The technique focuses on the problem related to weaving one or several aspects to a class. When integrated to control, aspects may affect the original behavior of classes. Knowing that classes collaborate to achieve their respective responsibilities, this may have an impact (in an indirect way) on the behavior of its client classes. The objective is to ensure that the integration is done correctly, without altering the original behavior of classes. The proposed approach is based on UML Statechart Diagrams of the classes under test and the code of related aspects. The technique focuses on an aspect(s)-class block. It consists of generating, in a first step, the testing sequences corresponding to the different scenarios of the statechart diagram of the class under test. The integration of related aspects is done in a second step, in an incremental way. The testing sequences generated from the statechart diagram of the class are extended according to the behavior of related aspects.

The testing sequences are generated automatically by the AJUnit tool. The tool, based on the model of JUnit, also supports the execution and verification of the testing sequences. We focus, in this paper, on AspectJ programs. Testing an aspect(s)-class block is done incrementally. Furthermore, the testing sequences (the corresponding code generated by AJUnit) are archived. In the case of a change instantiated on a class or on one of its related aspects, only the testing sequences corresponding to the affected parts of the program will be retested, avoiding in this way to retest the entire block. The testing effort (regression testing) following a change is reduced. The same approach is followed when introducing a new aspect in the code. The impact of this introduction on an existing aspect(s)-class block is identified. Furthermore, the adopted approach reduces the complexity of detecting eventual conflicts between aspects. We focus, in the context of our technique, on the conflicts that can appear following the integration of several aspects to a same class. The technique is based on several testing criteria that we defined. The



generation and verification process of the testing sequences is completely automated. The technique and associated tool are illustrated using two case studies.

The remainder of the paper is organized as follows. In Section 2, we give a brief overview of related work. Section 3 presents some basic concepts of AspectJ. In Section 4, we present the testing criteria we defined. The technique we propose is described in Section 5. Section 6 gives an overview of the testing tool AJUnit that we developed. Section 7 illustrates the tool using two case studies. Finally, Section 8 gives a general conclusion and some future work directions.

2 RELATED WORK

Alexander et al. discussed in [Alex 02] the challenges of the aspect paradigm and mentioned the importance of developing specific testing techniques. They also discussed in [Alex 04] the different sources of faults in an aspect-oriented program. They proposed a model that includes six types of faults. Ubayachi and Tamai [Ubay 02] proposed a model checking based technique. Aspects are classified according to whether they modify or not the state of the system. Zhao presents in [Zhao 02, Zhao 03] a method for unit testing of classes and aspects based on control flow graphs. Xu et al. proposed different approaches for testing aspect-oriented programs [Xu 05a, Xu 05b, Xu 05c, Xu 06]. They presented in [Xu 05a] a technique based on statechart diagrams. It consists, in fact, on an extension of the FREE method (Flattened Regular Expression [Bind 00]) developed for object-oriented programs. The technique considers aspect-class sets as a unique block. However, if one of the elements (class or aspect) is modified, or if an aspect influencing the class is added, the entire block is retested. In [Xu 05b], they presented a unit testing technique based on control flow. The model used is called Aspect Scope State Model (ASSM). They also proposed in [Xu 05c] an approach based on different models (class diagrams, aspect diagrams and sequence diagrams) to produce test cases covering the interactions between aspects and classes. Mahoney et al. presented in [Maho 04] a code generation framework that allows managing transversal concerns using statechart diagrams. Mortensen et al. presented in [Mort 04] several testing criteria. Their approach combines two traditional testing techniques (structural and mutation testing). This technique consists primarily on identifying the faults related to the code introduced by an advice. A similar approach, but more complete, was proposed by Deursen et al. in [Deur 05]. In their testing strategy, they used the fault model proposed by Alexander et al. [Alex 04]. Zhou et al. [Zhou 04] proposed a unit testing technique based on the source code of a program. Sere [Sere 03] proposed a technique based on static analysis of aspects. The approach is based on a syntactic model of pointcut indicators using regular expressions. Xie et al. proposed in [Xie 05] a framework for the automatic generation of unit tests using AspectJ compiled code (bytecode).

3 ASPECTJ : BASIC CONCEPTS

AspectJ represents a seamless aspect-oriented extension of Java [Ajws 05, Zhao 04]. Eclipse (with AJDT) [Ajpg 02] is a compiler as well as a platform supporting the development of AspectJ programs. AspectJ achieves modularity with aspect abstraction mechanisms, which encapsulate behavior and state of a crosscutting concern. It introduces several new language constructs such as *introductions, jointpoints, pointcuts* and *advice*. Aspects typically contain new code fragments that are introduced to the system. Aspects make it explicit where and how a concern is addressed, in the form of jointpoints and advice. Aspects execution depends upon context (control and data flow) provided by the core concerns represented by classes signature [Redd 06]. Moreover, aspects have the possibility to make significant changes to the semantics of a core concern [Mcea 05]. An aspect gathers pointcuts and advice to form a regrouping unit [Ajws 05, Balt 01, Xie 05]. An aspect is similar to a Java or C++ class in the way that it contains attributes and methods [Zhao 04]. The essential mechanism provided for composing an aspect with other classes is called *joint point*. Even more, join points are well-defined points in the execution in a program [Kicz 01]. AspectJ makes it possible to define joint points in relationship to a method call or a class constructor. A pointcut is a set of joint points and aims of referring certain values at those joint points [Kicz 01]. A pointcut can be built out of other pointcuts using logical operators (and, or, and not) [Masu 03]. AspectJ includes a variety of primitive pointcut designators that identify join points in different ways. An advice is a method like abstraction used to specify the code to execute when a jointpoint is reached. It can also expose some of the values in the execution of a jointpoint. Pointcuts and advice define integration rules. For more details see [Ajpg 02, Ajws 05].

4 TESTING CRITERIA

A testing criterion is a rule or a set of rules imposing conditions on a testing strategy [Offu 99b, Abdu 00, Mort 04, Xie 05]. It also specifies the required tests in terms of identifiable coverage of the program specification used to evaluate a set of test cases (also known as test suite) [Mort 04]. Testing criteria are used to determine what should be tested without telling how to test it. Testing engineers use these criteria to measure the coverage of a test suite [Offu 96, Wuye 02]. They are also used to evaluate the quality of a test suite. In this section, we present several testing criteria. Classic testing criteria such as those presented in [Offu 99a, Offu 99b, Offu 03, Viei 00, Bria 04] support the generation of testing sequences from classic statechart diagrams of classes. As mentioned previously, aspects have the capacity of affecting the behavior of classes. To cover the new dimensions introduced by aspects, we must develop new testing criteria. The testing

technique we propose is incremental. We suppose that the class is tested separately in a first step. We are then interested in testing the extended class while integrating one or several aspects. The adopted technique allows identifying the sequences of the statechart diagram of the class affected by the aspects.

Transition Coverage Criterion

In a UML Statechart Diagram, a transition represents a link between two states. Each transition affected or created by an aspect must be tested at least once. According to [Offu 99b], a tester should also test each pre-condition of the specification at least once to make sure that it will always be possible to execute the given scenario. A test covers a transition only when the corresponding pre-condition is true.

C1: Each transition affected by an aspect must be tested at least once.

We consider in what follows an example of a simple statechart diagram of a *Stack* class. Figure 1 shows the case where a method is affected after the integration of an aspect. This example shows the case of method *Trace* of the aspect that is executed after method *Push* of the *Stack* class. To avoid overloading the diagram, the *after* advice is only represented on the transition *Push* between states *Stack empty* and *Stack not full*. The set of *Push* transitions of the diagram are then concerned by this testing criterion and must be retested:

(1) Push from state Stack empty to Stack not full,
(2) Push from state Stack not full to Stack not full and
(3) Push from state Stack not full to Stack full.

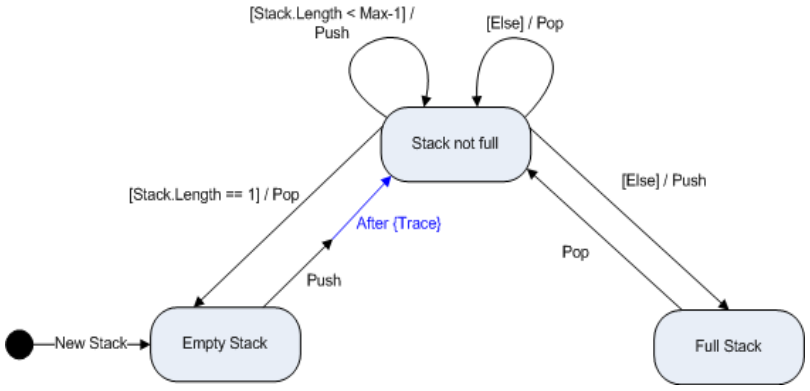


Figure 1 : Statechart diagram with an after advice on the Push method.

Sequence Coverage Criterion

A sequence corresponds to a logical suite of several transitions. It represents a particular scenario that can be executed at least once. When a class is affected by one or several aspects, its behavior can be altered. It is then imperative to adequately test, in particular,

all the sequences affected by aspects. In the previous example, all the sequences, including the affected or created transitions must then be tested again at least once.

C2: All the affected sequences by one or several aspects must be retested.

In the previous example, the set of transitions that include at least one *Push* transition must be retested:

(1) Stack empty–*Push* → Stack not full–*Pop* → Stack empty,

(2) Stack empty–*Push* → Stack not full–*Push* → Stack not full–*Pop* → Stack not full–*Pop* → Stack empty,

(3) etc.

Advice Execution Coverage Criterion

This criterion concerns the around-type advice. When a method of a class is affected by an advice of that type, every possible execution of the advice must be considered. In Figure 2, for example, we can have an *around* advice on method *Push* that will allow the execution of the method only if the element to be added do not exist in the stack. We must then test the advice with the two possible cases that can occur: when the element exists in the stack and when it does not exist. Furthermore, the set of sequences of the diagram including the affected transition (with the set of execution combinations of the advice) must be retested at least once.

C3: When a transition is affected by an advice, test at least once the set of possible executions of the advice in each sequence.

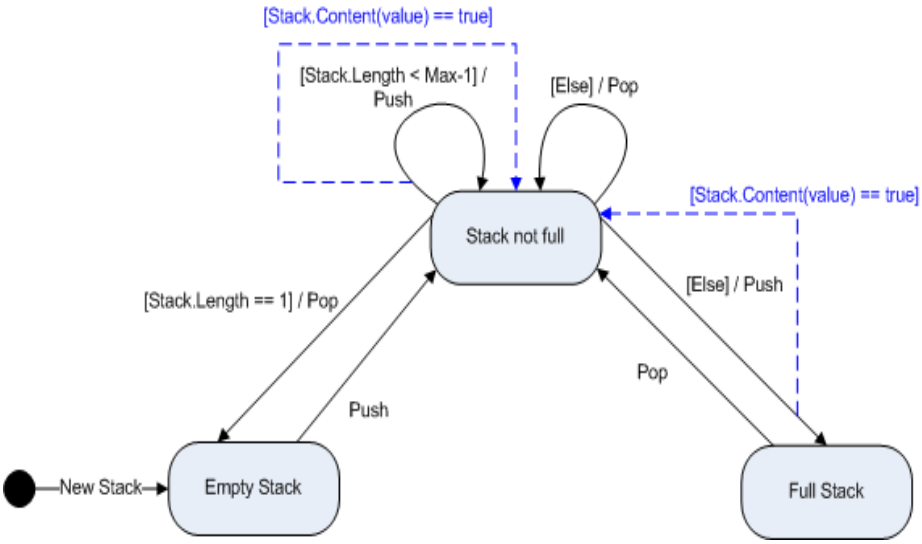
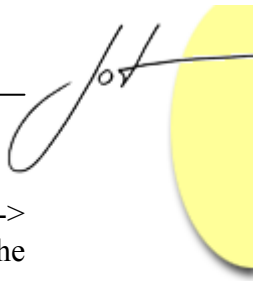


Figure 2**:** Statechart diagram with an around advice on method Push.



In the example, only the transitions Stack not full –> Stack not full and Stack not full -> Stack full are concerned. According to the criterion *C3*, we must consider all the possibilities. We then have the following sequences:

(1) Stack empty–*Push* → Stack not full–*Push a present object* → Stack not full–*Pop* → Stack empty

(2) Stack empty–*Push* → Stack not full–*Push a new object* → Stack not full–*Pop* → Stack not full–*Pop* → Stack empty

(3) Stack empty–*Push* → Stack not full–*Push a new object* → Stack not full– *Push a present object* → Stack not full–P*op* → Stack empty

(4) Stack empty–*Push* → Stack not full–*Push a new object* → Stack not full–*Push a new object* → Stack full–*Pop* → Stack not full–*Pop* → Stack empty

etc.

Multi-Aspect Integration Coverage Criterion

Several aspects can be weaved to a same method of a class. In this situation, conflicts may arise. In spite of certain mechanisms making it possible to specify the execution order, it is always possible to be confronted to a random sequencing. The context is important since executing an aspect before another can change the state of the system.

C4: If a method is affected by several aspects, the sequences that include that method must be retested at least once by covering all possible advice permutations.

5 AUTOMATED TESTING SEQUENCES GENERATION

The proposed technique supports the generation of testing sequences basically from the UML statechart of the class under test and the behavior of its related aspects extracted by static analysis of the source code (AspectJ). The UML statecharts of the classes and the behavior of related aspects are described using XML. The technique consists of generating from the original statechart of the class under test, an Extended StateChart (ESC) representing the dynamic behavior of the class extended with the integration of related aspects. The ESC model illustrates the behavior of the class extended with weaved aspects. The testing sequences are generated, from the ESC model, according to the testing criteria defined in Section 4. The primary objective is to verify that the original behavior of the class is not altered by aspects, and to insure that aspects behave correctly. Figure 3 illustrates the major steps of the testing sequences generation process. A single aspect is considered in this case.

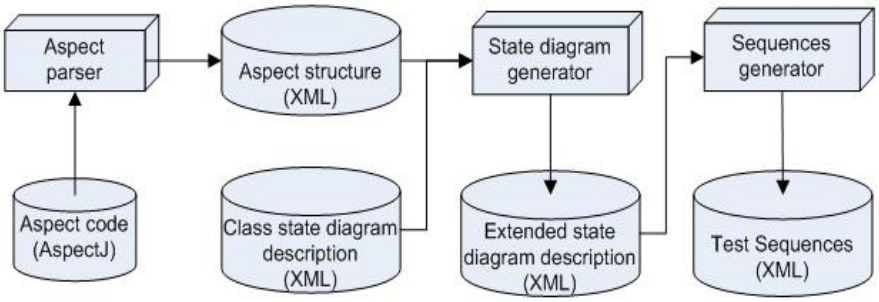


Figure 3: Testing sequences generator: Architecture.

The first step of the technique consists of analyzing (static analysis) the aspect's code and generating an XML model describing its behavior. Only the aspect's parts related to the class under test are considered. The next step consists of analyzing jointly the XML descriptions of both the statechart of the class and its related aspect. The result of this analysis consists of a new XML description corresponding to the extended statechart.

```
public class Stack
  {
    private Node _firstNode;
    public Stack() { this._firstNode = null; }
    public Node getFirstNode() { return this._firstNode; }
    public void setFirstNode(Node node)
    { this._firstNode = node; }
    public void Push(Node node)
    {
          node.setNextNode(this._firstNode);
          this._firstNode = node;
    }
    public Node Pop()
    {
          Node popNode = this._firstNode;
          this._firstNode = popNode.getNextNode();
          return popNode;
    }
  }
```

Figure 4: Code of the class *Stack*.

```
public aspect StackAspect
{
  private boolean Stack.Content(Node node)
  {
            Node currentNode = this.getFirstNode();
            while(currentNode != null)
            {
                if (currentNode.equals(node))  return true;
                currentNode = currentNode.getNextNode();
            }
            return false;
  }
  pointcut StackPublicMethod() : call( public * *(..)) && target(Stack);
   before () : StackPublicMethod()
       {System.out.println("Enter in method : " + thisJoinPoint.getSignature().toString()); }
    void around ( Node node, Stack stack ) :
            call(public void Stack.Push(Node))
            && args(node) && target(stack)
               { if ( !stack.Content(node) ) proceed(node, stack); }
     after () : StackPublicMethod()
        { System.out.println("Exit of method : " + thisJoinPoint.getSignature().toString()); }
}
```

Figure 5: Code of the aspect *StackAspect*.

We identify the impact of the aspect on the behavior of the class. The potentially affected scenarios in the behavior of the class, following the integration of the aspects, are identified. The final step corresponds to the generation of testing sequences from the XML description of the extended statechart according to the defined testing criteria. We illustrate, in what follows, the major steps of the technique. Figure 4 gives the code for a Stack class, and figure 5 gives the AspectJ code for the considered *StackAspect* aspect. This aspect includes three advice. The *before* and *after* advice are used to trace the execution of each public method of the class. The *around* advice allows avoiding to push in the stack an already existing element. The aspect also contains a private introduction that allows determining if a stack contains a particular element. Figure 6 presents the original statechart of the *Stack* class, and figure 7 presents its extension after the integration of aspect *StackAspect*.

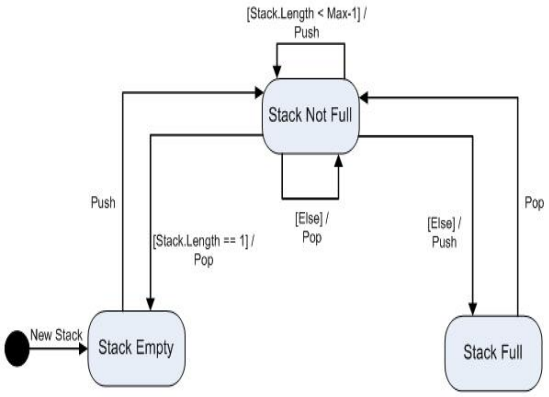


Figure 6: Original statechart diagram.

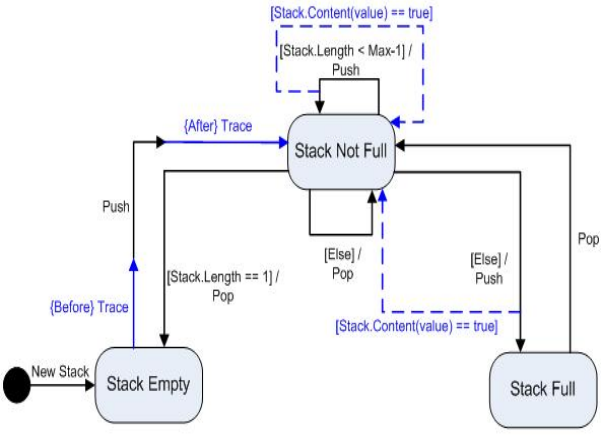


Figure **7:** Extended statechart diagram.

The testing process begins by testing the class without any aspect. This is done to reduce the complexity of the testing process and to eliminate the eventual faults related to the object code. After that, the technique consists of integrating incrementally the aspects to the class. In this step, we identify the methods of the class that are affected by the integration of an aspect and determine at the same time which transitions (and scenarios) are affected in the statechart. In the extended statechart of the class S*tack* given in figure 7, we can see that conditions on method *Push* have been added by the *around* advice of the aspect. Advices *before* and *after* are executed respectively before and after methods *Push* and *Pop* of the class. In order to avoid overloading the diagram, we deliberately omitted some of details related to transitions *Push* and *Pop*. The third step consists of transforming the extended statechart in a tree (given in figure 8) and generating from the tree the adequate testing sequences.

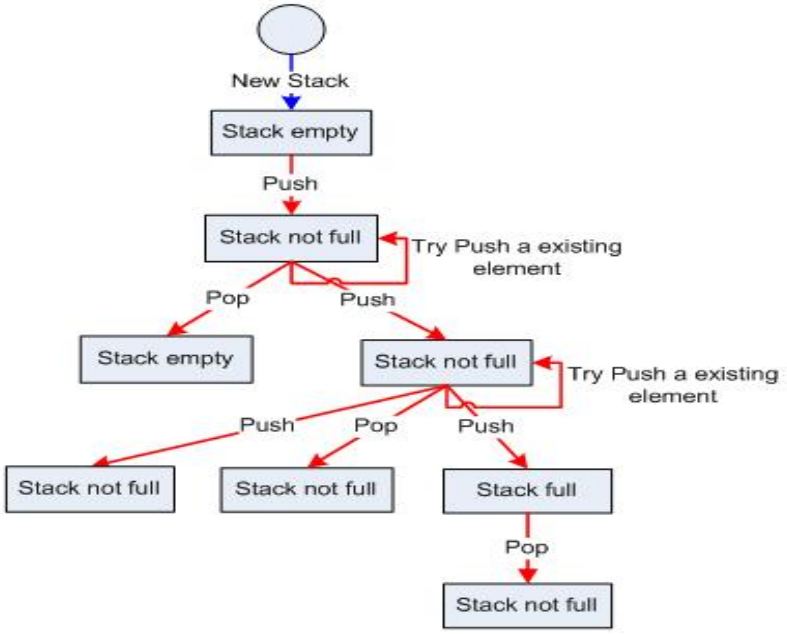


Figure 8: Sequences Tree.

The *Contains* method is not shown in the tree because it does not affect the behavior of the class. The methods of the class that are affected by the aspect and those introduced are represented in figure 8. Those methods have to be tested. Moreover, it is possible in some cases that the integration of aspects creates new states in the statechart of the class under test. The new sequences including the new states have to be considered. From the example, we can generate the following set of testing sequences:

(1) New Stack, Push, Pop,

(2) New Stack, Push, Try Push an existing element, Pop,

(3) New Stack, Push, Push, Push until stack is full, etc.

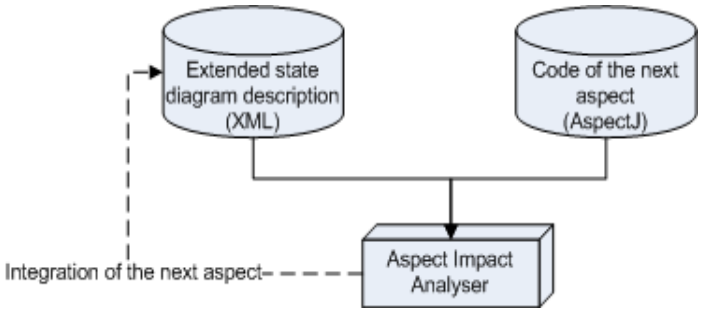


Figure 9: Incremental integration of aspects.

The adopted approach is iterative. In the case where several aspects are related to a class, the method consists of integrating one at a time, beginning with the most complex. The choice of an aspect during an iteration is based on criteria such as the intrinsic complexity of the aspect and its coupling to the considered class. The test of an aspect-class block is done incrementally, according to the generated sequences. The extended statechart of the class under test, in the case where several aspects are weaved, is constructed iteratively as illustrated in figure 9. Moreover, and as mentioned previously, the testing sequences are archived. In the case of a change instantiated whether on the class or on one of its related aspects, the previous testing sequences will be reused. The impact of the change on the testing sequences will be identified. Only the affected sequences (directly or indirectly) will be retested. This will prevent the retesting of the entire block (aspect(s)-class) as it is the case in the approach defined by Xu et al. in [Xu 05]. We assume that the followed approach allows reducing the testing effort. Furthermore, and over the testing sequences generation and verification process, the incremental integration will offer the possibility of a better concentration and the possibility to identify the eventual conflicts between aspects. The localisation of eventual errors will then be facilitated.

As illustrated in figure 9, the XML description of the statechart of the class under test is replaced during the testing process by the XML description of the class extended by adding an aspect. Therefore, when a class is affected by several aspects, we can test this class incrementally by progressively including aspects. Furthermore, the incremental integration of aspects, with the analysis of the different retained descriptions, allows us to identify the result of each extension of the class (integration of an aspect) and, at the same time, better guide the testing process and identify the eventual conflicts between aspects. The major steps of our method are described in the following algorithm:

1. Generating the unit testing sequences for the class.
2. Testing the class separately.
3. Introducing aspects. As long as there are aspects not integrated
4. Introducing an aspect.
    a. Generating the extended statechart of the class
    b. Identifying the affected transitions by the integration
    c. Constructing the tree corresponding to the extended diagram
    d. Generating the testing sequences affected or created by the aspect
    e. Testing the class with the integrated aspect
    f. If there is no problem encountered, return to step 4
5. End.

6 AJUNIT: A SUPPORTING FRAMEWORK

AJUnit extends the Java unit testing framework JUnit. For a given class, it automatically generates appropriate unit testing classes according to the iterative method described in Section 5. The tool executes the generated tests and produces results. Figure 10 shows the major steps of the process.

AJUnit includes several components. An AspectJ analyzer, as an Eclipse plug-in (1), generates an XML model from the code of an aspect describing its behavior related to the class under test. An analyzer of the impact of the integration of the aspect to the class (2) unifies the XML descriptions of the statechart diagram of the class under test and the introduced aspect. It produces the XML description corresponding to the ESC model of the class. This model integrates the dependencies of each advice and pointcut of the aspect on the class. The goal is to identify the sequences that must be tested. The sequences generator (3), starting from the ESC model, generates the testing sequences according to the method described in Section 5. The generated tests cover the impact of the behavior of the aspect on the class. The last component (4), based on JUnit, generates the code of the appropriate test classes and verifies, such as JUnit, their execution. The extension of JUnit is essentially based on the creation of two classes TestSequenceCase and TestSequenceSuite, derived respectively from TestCase and TestSuite (see Figure 11).

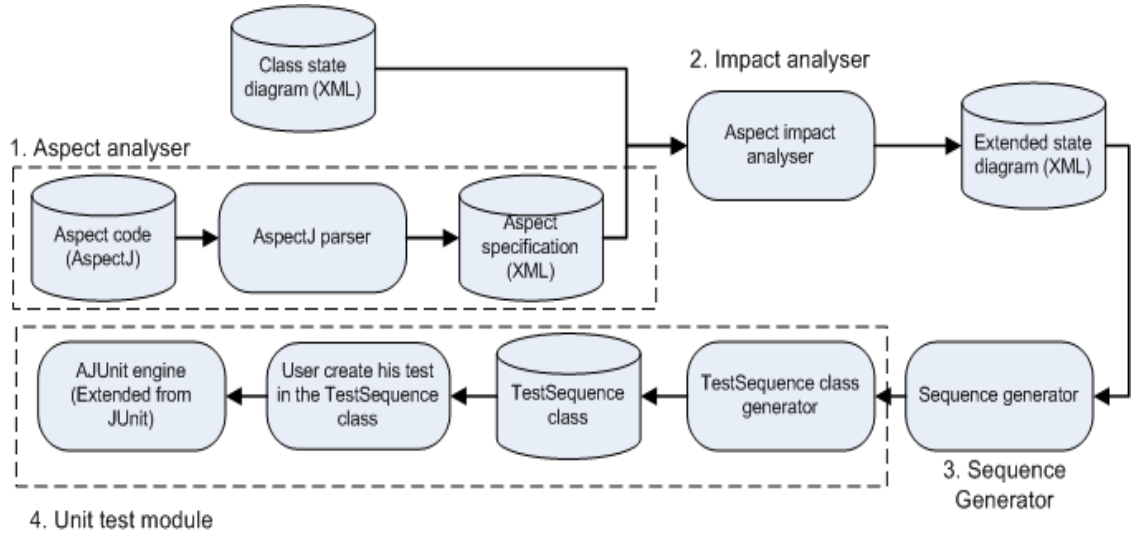


Figure 10: Testing Sequences identification.

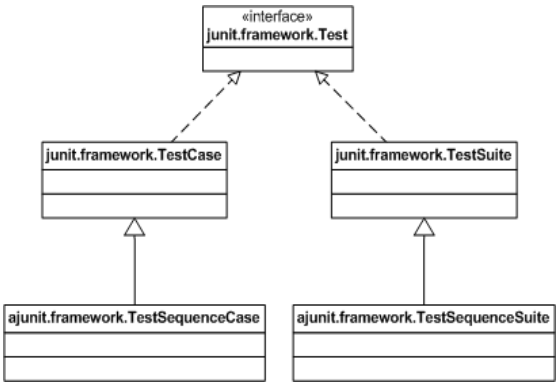


Figure 11 : AJUnit's UML diagram.

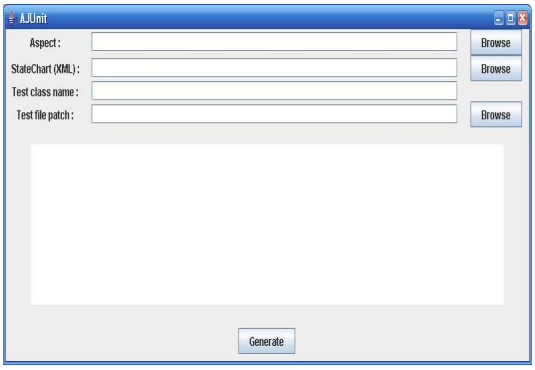


Figure 12 : Tool.

Figure 12 presents a screen capture of the AJUnit tool. From this screen, the user can select the file containing the aspect, the file containing the XML description of the statechart diagram of the class, the name of the test class which will be generated, as well as the directory in which to create the file.

7 CASE STUDIES

We present, in this section, two case studies. The first one shows the case of a unique aspect weaved to a class. The second one considers the case where two aspects are integrated to a class.

Case Study : One Aspect – One Class

We consider for this case study a class that offers a search service in a knowledge data base to a group of authenticated users. The class contains the following methods: *Connect, Login, GotoSearch, Search* and *Logout*. A user must enter his user name and password to be able to perform searches. At every step, the user can close his work session. We also consider an aspect that allows keeping a history of unsuccessful connection attempts, the possibility of saving a user's logout into a journal as well as the search requests that lead to empty results. The statechart diagram of the class and its code are given respectively in Figures 13 and 14.

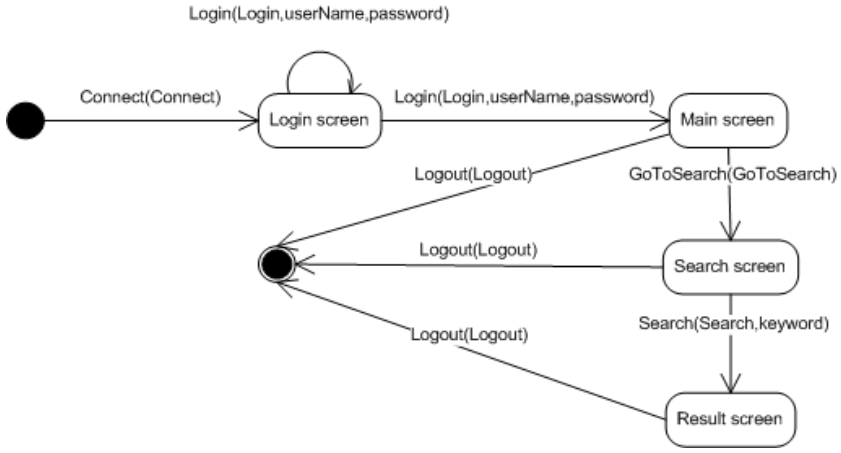


Figure 13: Statechart Diagram of the class.

```
public class Example1
{
   public static int Disconnected = 0;
   public static int Connected = 1;
   public static int Logging = 2;
   public static int Logged  = 3;
   public static int Searching = 4;
   private int m_state;
   public Example1()  {  m_state = Disconnected; }
   public int GetState() {  return m_state; }
   public void Connect() {  m_state = Connected; }
   public boolean Login(String username, String password)
   {
      m_state = Logging;
      if(username.equals(password)) { m_state = Logged;}
      else {  m_state = Connected; }
     return m_state == Logged;
   }
  public void Logout() {  m_state = Disconnected; }
  public String Research(String keyword)
   {
     if(m_state == Logged)  { return Search(keyword); }
     else { return null; }
   }
   public void GoToSearch()  { //Navigate to the page... }
   private String Search(String keyword)
   {
      m_state = Searching;
      //Sreaching...
      m_state = Logged;
      return "result";
   }
 }
```

Figure 14: Source code of the class.

```
public aspect Example1Aspect


{
  pointcut OnLogout() : call(public void Logout()) && target(Example1);
  pointcut OnLogin(String username, String password) : call(public boolean Login(String, String))
            && target(Example1)
            && args(username, password);
  pointcut OnResearch(String keyword) : call(public String Research(String))
            && target(Example1)
            && args(keyword);
  before() : OnLogout () { System.out.println("Before Logout"); }
  after(String username,String password) returning (boolean result): OnLogin(username, passwd)
    {
      if (!result) System.out.println("Login fail : " + username + ", " + password);
    }
    after(String keyword) returning (String researchResult) : OnResearch(keyword) {
      if(researchResult == null) System.out.println("Unautorised research on : " + keyword);
    }
}
```

Figure 15: Aspect's code.

The considered aspect includes three advice (see Figure 15) that all write an element in the event journal. The first advice, of *before* type, is linked to the *OnLogout* pointcut and allows the inscription of the disconnection of the user into the journal. The second advice, of *after* type, is linked to the *OnLogin* pointcut. This advice leaves a trace of execution only if the user's authentication fails. The last advice, of *after* type, allows tracing the failed search requests. Figure 16 shows the tree that will be generated following the execution of the AJUnit tool with the targeted methods by the aspect marked in red.

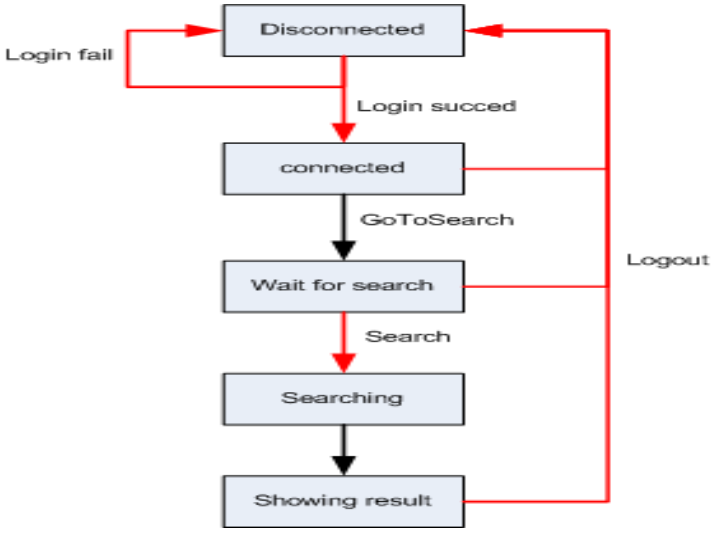
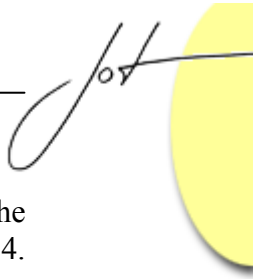


Figure 16: Sequences' Tree.

```
public class Exemple1TestSequence extends TestSequenceCase
{
 public static void main(String[] args)
   {ajunit.sequenceTextui.TestRunner.run(Exemple1TestSequence.class);}
 public void testLogin() { }
 public void testResearch() { }
 public void testLogout() { }
 public void testConnect() { }
 public void testGoToSearch() { }
 public void sequence1()
  { testConnect(); testLogin(); testLogin(); testGoToSearch(); testResearch(); testLogout(); }
 public void sequence2()
   { testConnect(); testLogin(); testLogin(); testGoToSearch(); testLogout(); }
 public void sequence3()
   { testConnect(); testLogin(); testLogin(); testLogout(); }
 public void sequence4()
   { testConnect(); testLogin(); testGoToSearch(); testResearch(); testLogout(); }
 public void sequence5()
   { testConnect(); testLogin(); testGoToSearch(); testLogout(); }
 public void sequence6()
   { testConnect(); testLogin(); testLogout(); }
}
```

Figure 17: Resulting test class.



As we can see by examining the code of the aspect and the statechart diagram of the class, five transitions must be tested according to the first test criterion of Section 4. Three of these transitions are of the Logout category for the first advice and the two others are two possible paths of the Login method covered by the second advice. If we refer to the second test criterion (Section 4), six testing sequences have to be covered. There are two sequences for each of the three Logout transitions, one if the authentication of the user succeeds the first time, and one other when it is not the case. The generated code corresponds to the sequences given in Figure 17. With the performed analysis, the tool extracts from the statechart of the class the sequences that must be tested. From these sequences, the tool generates a test class (Figure 18) that contains a method for each of the identified sequences, as well as a test method for each of the methods included in these sequences.

```
public class Exemple2TestSequence extends TestSequenceCase
{
 public static void main(String[] args) { ajunit.sequenceTextui.TestRunner.run(Exemple2TestSequence.class); }
 private Exemple2 m_class;
 public void setUp() {  m_class = new Exemple2(); }
 public void tearDown() {  m_class = null; }
 public void testConnect() {assertEquals(Exemple2.Disconnected ,m_class.GetState());  m_class.Connect();
                            assertEquals(Exemple2.Connected ,m_class.GetState()); }
 public void testResearch() {assertEquals(Exemple2.Logged, m_class.GetState());  assertEquals("result",
                             m_class.Research("keyword")); assertEquals(Exemple2.Logged ,m_class.GetState());  }
 public void testLoginSucces() {assertEquals(Exemple2.Connected ,m_class.GetState());
                                m_class.Login("succes", "succes");
                                assertEquals(Exemple2.Logged, m_class.GetState());  }
 public void testLoginFail() {assertEquals(Exemple2.Connected ,m_class.GetState());
                              m_class.Login("login will","fail");
                              assertEquals(Exemple2.Connected, m_class.GetState()); }
 public void testGoToSearch() {assertEquals(Exemple2.Logged ,m_class.GetState());  m_class.GoToSearch();
                               assertEquals(Exemple2.Logged ,m_class.GetState()); }
 public void testLogout()  {m_class.Logout();  assertEquals(Exemple2.Disconnected, m_class.GetState());  }
 public void sequence1()  {System.out.println("sequence 1");   testConnect(); testLoginFail(); testLoginSucces();
                           testGoToSearch(); testResearch();  testLogout(); }
 public void sequence2() {System.out.println("sequence 2");  testConnect(); testLoginFail(); testLoginSucces();
                          testGoToSearch(); testLogout(); }
 public void sequence3() {System.out.println("sequence 3");  testConnect(); testLoginFail(); testLoginSucces();
                          testLogout();  }
 public void sequence4() {System.out.println("sequence 4");   testConnect(); testLoginSucces(); testGoToSearch();
                          testResearch(); testLogout(); }
 public void sequence5() {System.out.println("sequence 5"); testConnect(); testLoginSucces(); testGoToSearch();
                          testLogout(); }
 public void sequence6() {System.out.println("sequence 6"); testConnect(); testLoginSucces(); testLogout(); }
}
```

Figure 18: Completed test class.

After that, the user will be able to complete, as it is the case for JUnit, his test class. Figure 18 gives an example of a completed class. Furthermore, knowing that the class was tested before without the integration of the aspect (with JUnit), the user could reuse the unit tests already performed for each method to complete his class (and its methods, the ones beginning with test such as *testLogin*). The result of the execution is given in

Figure 19. The figure shows the six tested sequences while also considering the selected testing criteria. With the AJUnit tool, we can precisely target the sequences to be retested. This allows not retesting every parts of the program (as mentioned in Section 5), as well as the significant reduction of testing efforts. Furthermore, the user is assisted during the entire testing process.

```
.sequence 1
Login fail : login will, fail
Before Logout
.sequence 2
Login fail : login will, fail
Before Logout
.sequence 3
Login fail : login will, fail
Before Logout
.sequence 4
Before Logout
.sequence 5
Before Logout
.sequence 6
Before Logout

Time: 0,031

OK (6 tests)
```

Figure 19: Sequences' execution.

Case Study : Two Aspects – One Class

The second case study presents the case where two aspects are weaved to a class. The aspects, as mentioned in Section 5, will be integrated incrementally with the objective to reduce the complexity of the test and to facilitate error detection. We adopted an iterative strategy that consists on integrating the more complex aspect first. The class used for the case study is a parsing class for which the data source can be a local file or a file accessible through a network. Figure 20 gives the statechart diagram of this class.

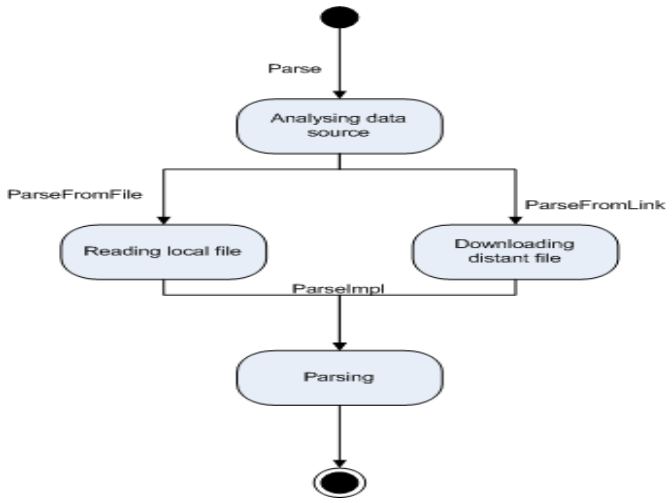


Figure 20: Statechart diagram of the class.

The first aspect that will be integrated, for which the code is given in Figure 21, has as role to insure that the version of the local file is identical to the one found on the server to make sure to avoid working on an obsolete version.

```
public aspect SourceControlAspect
{
    pointcut FileOpen(String filename) : call(public void ParseFromFile(String))
              && target(Example3)
              && args(filename);

    before(String filename) : FileOpen(filename)
    { SourceControlMock.SetVersion(SourceControlMock.GetVersion() + 1); }
}
```

Figure 21: Source of the control aspect code.

```
public class Example3WithSourceControlTestSequence extends TestSequenceCase
{
 public static void main(String[] args)
 {
  ajunit.sequenceTextui.TestRunner.run(Example3WithSourceControlTestSequence.class);
 }
 public void testParse() { }
 public void testParseImpl() { }
 public void testParseFromLink() { }
 public void testParseFromFile() { }
 public void sequence1() {  testParse();  testParseFromFile();  testParseImpl(); }
}
```

Figure 22: Test class.

As we can see, the aspect is linked to method *ParseFromFile* of the class. The test class generated with AJUnit is given in Figure 22. There is only one testing sequence in this case. The second aspect, for which the code is given in Figure 23, has as role to insure

file security for the files downloaded from the network. It is linked, in particular, to the *ParseFromLink* method. As mentioned previously, the integration process is performed incrementally (see Figure 9). In this case, the statechart diagram (XML – ESC description) that will be considered as input will correspond to the one obtained after the integration of the first aspect. It will then be extended by the integration of the second aspect. The AJUnit tool will use the new extended diagram (XML – ESC description) resulting from the integration of the second aspect for generating the testing sequences.

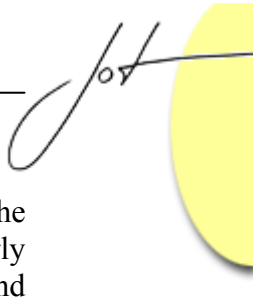
```
public aspect VirusCheckAspect
{
   pointcut Download(String filename) : call(public void ParseFromLink(String))
            && target(Example3)
            && args(filename);

   void around(String link) throws Exception: Download(link)
   {
      if(link.indexOf("virus") != -1)  {  throw new Exception(); }
      Else  {  proceed(link); }
   }
}
```

Figure 23: File Security Verification aspect code.

```
import ajunit.framework.TestSequenceCase;
public class Example3WithBothTestSequence extends TestSequenceCase
{
   public static void main(String[] args)
   {ajunit.sequenceTextui.TestRunner.run(Example3WithBothTestSequence.class); }
   private Example3 m_instance;
   public void setUp()  {  m_instance = new Example3(); }
   public void testParseImpl() { }
   public void testParse() { }
   public void testParseFromFile()
   {  SourceControlMock.SetVersion(4);
      m_instance.ParseFromFile("fileName");
      assertEquals(5, SourceControlMock.GetVersion()); }
   public void testParseFromLink()
   {  String correctLink = "secureLink";
      boolean exceptionThrow = false;
      try {  m_instance.ParseFromLink(correctLink); }
      catch (Exception e)  { exceptionThrow = true; }
      assertFalse(exceptionThrow);
      String wrongLink = "linkWithVirus";
      exceptionThrow = false;
      try {  m_instance.ParseFromLink(wrongLink); }
      catch (Exception e)  {  exceptionThrow = true; }
      assertTrue(exceptionThrow);  }
   public void sequence1() {  testParse();  testParseFromFile();  testParseImpl(); }
   public void sequence2() {  testParse();  testParseFromLink();  testParseImpl(); }
}
```

Figure 24: Test class' code including both aspects.



The obtained test class, given in Figure 24, shows that the sole sequence affected by the second aspect corresponds to the one presented by the sequence2 method. We can clearly see, through this example, the incremental process our technique provides. In this second test class, we can find all the affected sequences following the introduction of the first and second aspect. Furthermore, our technique facilitates the identification of errors relative to the integration of an aspect, but also the ones relative to eventual conflicts between aspects. To proceed to the test of the whole set of sequences, the user will complete the Example3WithBothTestSequence class (code in bold). This code corresponds to the test class (with JUnit) before the integration of aspects. If we execute the test class without implementing the aspects, we obtain the following results, which shows that both testing sequences failed.

```
There were 2 failures:
1)  sequence1(example.Example3WithBothTestSequence)junit.framework.ComparisonFailure:
    expected:<...+1> but was:<...>
at example.Example3WithBothTestSequence.testParseFromFile
                                                (Example3WithBothTestSequence.java:27)
at example.Example3WithBothTestSequence.sequence1
                                                (Example3WithBothTestSequence.java:56)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
at ajunit.sequenceTextui.TestRunner.run(TestRunner.java:48)
at example.Example3WithBothTestSequence.main(Example3WithBothTestSequence.java:8)
2) sequence2(example.Example3WithBothTestSequence)junit.framework.AssertionFailedError
at example.Example3WithBothTestSequence.testParseFromLink
                                                (Example3WithBothTestSequence.java:52)
at example.Example3WithBothTestSequence.sequence2
                                                (Example3WithBothTestSequence.java:61)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
at ajunit.sequenceTextui.TestRunner.run(TestRunner.java:48)
at example.Example3WithBothTestSequence.main(Example3WithBothTestSequence.java:8)

FAILURES!!!
Tests run: 2,  Failures: 2,  Errors: 0
```

On the contrary, both testing sequences are executed without any error:

```
OK (2 tests)
```

8 CONCLUSIONS AND FUTURE WORK

We presented, in this paper, a state-based unit testing technique for aspect-oriented programs and associated tool (called AJUnit). The technique is based on the dynamic behavior of classes and related aspects. It supports both the generation and verification of testing sequences. The technique focuses on the problem related to the weaving of one or several aspects to a class. The objective is to ensure that the integration is done correctly, without altering the original behavior of the classes. The technique focuses on an aspect(s)-class block.

The testing sequences are generated automatically by the AJUnit tool. Based on the model of JUnit, AJUnit also supports the execution and verification of the testing sequences. We focus, in this paper, on AspectJ programs. The adopted strategy is iterative. It consists on testing, in a first step, the class separately by using the JUnit tool. The integration of aspects is done in a second step. The generated sequences for testing the class are, in fact, extended incrementally according to the behavior of related aspects.

Furthermore, AJUnit was designed to support regression testing. The generated testing sequences (the corresponding code generated by AJUnit) are archived. The testing sequences are reused in the case of a change instantiated on the class or on one of its related aspects. The testing sequences corresponding to the affected parts of the aspect(s)-class block are identified and retested. This allows avoiding retesting all the aspect(s)-class block. The same approach is followed when introducing a new aspect in the code. The impact of this introduction is identified. Furthermore, the adopted approach reduces the complexity of detecting eventual conflicts between aspects.

As future work, we plan to experiment the developed framework on real AspectJ programs.

AKNOWLEDGEMENTS

This work was supported by a NSERC (Natural Sciences and Engineering Research Council of Canada) grant.

REFERENCES

[Abdu 00] A. Abdurazik and J. Offutt: “Using UML Collaboration Diagrams for Static Checking and Test Generation”, In Proceedings of the 3rd Internationl Conference on The Unified Modeling Language (UML ’00), York, UK, October 2000.

[Alex 02] Roger T. Alexander and James M. Bieman: “Challenges of Aspect-Oriented Technology”, In Proceedings of ICSE 2002 Workshop on Software Quality, 2002.

[Alex 04] Roger T. Alexander, James M. Bieman and Anneliese A. Andrews: “Towards the Systematic Testing of Aspect-Oriented Programs”, Department of Computer Science, Colorado State University, Fort Collins, Colorado, USA. Technical Report CS-4-105, March 2004.

[Ajpg 02] AspectJ, The AspectJ™ Programming Guide, 2002.

[Aosd 05] Aspect-Oriented Software Development Web Site (AOSD), http://.aosd.net/

[Ajws 05] AspectJ Web Site, http://eclipse.org/aspectj/

[Badr 05] M. Badri, L. Badri and M. Bourque-Fortin: "Generating Unit Testing Sequences For Aspect-Oriented Programs: Towards A Formal Approach Using UML State Diagrams", In Proceedings of the 3rd International Conference on Information & Communications Technology (ICICT 2005), Cairo, EGYPT5, 5, 6 December 2005.

[Ball 98] T. Ball: "On the Limit of Control Flow Analysis for Regression Test Selection", In Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA-98), vol. 23, 2 of ACM Software Engineering Notes, New York, March 1998.

[Balt 01] J. Baltus : « La Programmation Orientée Aspect et AspectJ : Présentation et Application dans un Système Distribué », Mini-Workshop: Systèmes Coopératifs, Institut d'informatique, Namur, 2001.

[Beiz 90] B. Beizer: "Software Testing Techniques", International Thomson Comuter Press, 1990.

[Bind 00] R. V. Binder: "Testing Object-Oriented Systems: Models, Patterns, and Tools", Addison-Wesley, 2000.

[Bria 04] L. C. Briand, Y. Labiche and Y. Wang: "Using Simulation to Empirically Investigate Test Coverage Criteria Based on Statecharts", In Proceedings of the ACM International Conference on Software Engineering (ICSE'04), pp. 86-95, Edinburgh, Scotland, UK, May 2004.

[Deur 05] A. Deursen, M. Martin and L. Mooned: "A systematic Aspect-Oriented Refactoring and Testing Strategy, and its Application to JhotDraw", In Proceedings of the 2005 ICSE Workshop on Modeling and analysis of concerns in software, St. Louis, Missouri, March 2005.

[Elra 01] Elrad, T., R.E. Filman, and A. Bader: "Aspect-oriented programming: Introduction", Communications of the ACM, 44(10): p. 29-32, 2001.

[JUnit 08] http://sourceforge.net/projects/junit/.

[Kicz 01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. Griswold: "An Overview of AspectJ", In Lecture Notes in Computer Science, Vo. 2072, p. 327-355, 2001.

[Lemo 04] Otávio Augusto Lazzarini Lemos, José Carlos Maldonado and Paulo Cesar Masiero: "Data Flow Integration Testing Criteria for Aspect-Oriented Programs", Primeiro Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspectos, 2004.

[Maho 04] Mark Mahoney, Atef Bader, Tzilla Elrad and Omar Aldawud: "Using Aspects to Abstract and Modularize Statecharts", In the 5th Aspect-Oriented Modeling Workshop In Conjunction with UML 2004 October 11-15, 2004.

[Masu 03] H. Masuhara and G. Kiczales: "Modeling Crosscutting in Aspect-Oriented Mechanisms", In Proceedings of ECOOP 2003, LNCS #2743, pp.2-28, 2003.

[Mass 07] P. Massicotte, L. Badri and M. Badri: "Towards a Tool Supporting Integration Testing of Aspect-Oriented Programs", In Journal of Object Technology, vol. 6, no. 1, January-February 2007.

[Mcea 05] N. McEachen and R. Alexander: "Distributing Classes with Woven Concerns. An Exploration of Potential Fault Scenarios", Proceedings of the 4th International Conference on Aspect-Oriented Software Development, pp 192-200, Chicago, Illnois, USA, March 14-18, 2005.

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

[Kand 00] Mohamed M. Kandé, Jörg Kienzle and Alfred Strohmeler: "From AOP to UML: Toward an aspect-oriented architectural modeling approach", In UML'2000 - The Unified Modeling Language: Advancing the Standard, York, UK, October 2-6, 2000.

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

[Offu 99a] Jeff Offutt and Aynur Abdurazik: "Generating Tests from UML Specifications", Second International Conference on the Unified Modeling Language (UML99), pages 416-429, Fort Collins, CO, October 1999.

[Offu 99b] Jeff Offutt, Yiwei Xiong and Shaoying Liu: "Criteria for Generating Specification-based Tests", In the 5th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '99), pages 119-131, October 1999.

[Offu 03] Jeff Offutt, Shaoying Liu, Aynur Abdurazik and Paul Ammann: "Generating test data from state-based specifications", In Journal of Software Testing, Verification and Reliability, 13(1):25-53, March 2003.

[Pawl 04] Renaud Pawlak and Housam Younessi: "On getting use cases and aspects to work together", Journal of Object Technology Vol. 3, No. 1, January-February 2004.

[Redd 06] Y.R. Reddy, S. Ghosh, R. France, G. Straw, J. Bieman, N. McEachen, E. Song, G. Georg: "Directives for composing aspect-oriented design class models", Trans. Aspect-Oriented Software Development, 2006.

[Sere 03] D. Sereni and O. de Moor: "Static analysis of aspects", In Proceedings of the 2nd International conference on aspect-Oriented Software development, March 2003.

[Sull 02] Kevin Sullivan, Lin Gu and Yuanfang Cai: "Non-Modularity in Aspect-Oriented Languages: Integration as a Crosscutting Concern for AspectJ", 2002.

[Suzu 99] Junichi Suzuki and Yoshikazu Yamamoto: "Extending UML with Aspects: Aspect support in the design phase", In ECOOP, 1999.

[Ubay02] N. Ubayashi and T. Tamai: "Aspect-oriented programming with model checking", In Proceedings of the 1st International Conference on Aspect-Oriented Software Development, 2002.

[Viei 00] Marlon E. Vieira, Marcio S. Dias and Debra J. Richardson: "Object-Oriented Specification-Based Testing Using UML Statechart Diagrams", In Proceedings of the Workshop on Automated Program Analysis, Testing and Verification, ICSE, 2000.

[Walk 99] R. Walker, E. Baniassad and G. Murphy: "An initial assessment of aspect-oriented programming", In Proceedings of the 21st International Conference on Software Engineering, Los Angeles, CA, May 1999.

[Wuye 02] Ye Wu, Mei-Hwa Chen and Jeff Offutt: "UML-based Integration Testing for Component-based Software", In Proceedings of the Second International Conference on COTS-Based Software Systems, September 2002.

[Xie 05] T. Xie, J. Zhao and D. Notkin: "Automated Test Generation for AspectJ Programs", In Proceedings of the AOSD '05 Workshop on Testing Aspect-Oriented Programs (WTAOP 05), Chicago, March 2005.

[Xudi 05a] Dianxiang Xu, Weifeng Xu and Kendall Nygard: "A State-Based Approach to Testing Aspect-Oriented Programs", In Proc. of the 17th International Conference on Software Engineering and Knowledge Engineering (SEKE'05), July 14-16, Taiwan.

[Xudi 05b] Dianxiang Xu, Weifeng Xu, Vivek Goel and Ken Nygard: "Aspect Flow Graph For Testing Aspect-Oriented Programs", Proceeding of the 8th IASTED International Conference on software Engineering and Applications. Oranjestad, Aruba (Caribbean), august 29-31, 2005.

[Xudi 05c] Dianxiang Xu, Weifeng Xu and Kendall Nygard: "A model-based approach to test generation for aspect-Oriented Program", AOSD '05 Workshop on Testing Aspect-Oriented Programs (WTAOP 05), Chicago, March 2005.

[Xu 06] Weifeng Xu and Dianxiang Xu: "State-Based Testing of Integration Aspects", In Proceeding of the second workshop on Testing Aspect-Oriented Programs (WTAOP 06), Juillet 2006, Maine, USA.

[Zaka 02] Aida Atef Zakaria, Hoda Hosny and Amir Zeid: "A UML Extension for Modeling Aspect-Oriented Systems", Second International Workshop on Aspect-Oriented Modeling with UML, 2002.

[Zhao 02] Jianjun Zhao: "Tool Support for Unit Testing of Aspect-Oriented Software", OOPSLA'2002 Workshop on Tools for Aspect-Oriented Software Development, Seattle, WA, USA, November 4, 2002.

[Zhao 03] Jianjun Zhao: "Data-flow-based unit testing of aspect-oriented programs", In Proc. 27th Annual IEEE International Computer Software and Applications Conference (COMPSAC'2003), pp.188-197. Dallas, Texas, USA, November 2003.

[Zhao 04] J. Zhao and B. Xu: "Measuring Coupling in Aspect-Oriented Systems", In 10th International Software Metrics Symposium (METRICS'2004), (Late Breaking Paper), Chicago, USA, September 14-16, 2004.

[Zhou 04] 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 (TECOS 2004), Net.ObjectiveDays, September 2004.

About the authors

Mourad Badri (Mourad.Badri@uqtr.ca) is professor of computer science at the Department of Mathematics and Computer Science of the University of Quebec at Trois-Rivières. He holds a PhD in computer science (software engineering) from the National Institute of Applied Sciences in Lyon, France. His main areas of interest include object and aspect-oriented software engineering, software quality attributes, and formal methods.

Linda Badri (Linda.Badri@uqtr.ca) is professor of computer science at the Department of Mathematics and Computer Science of the University of Quebec at Trois-Rivières. She holds a PhD in computer science (software engineering) from the National Institute of Applied Sciences in Lyon, France. Her main areas of interest include object and aspect-oriented software engineering, software quality attributes, maintenance, and web engineering.

Maxime Bourque-Fortin (Maxime.Bourque-Fortin@uqtr.ca) is a student of computer science at the Department of Mathematics and Computer Science of the University of Quebec at Trois-Rivières. He holds a master in computer science from the University of Quebec at Trois-Rivières. His main areas of interest include aspect-oriented programming and testing as well as various topics of software engineering.