JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2007**

Vol. 6, No. 1, January-February 2007

Towards a Tool Supporting Integration Testing of Aspect-Oriented Programs

Philippe Massicotte, University of Quebec at Trois-Rivières
Linda Badri, University of Quebec at Trois-Rivières
Mourad Badri, University of Quebec at Trois-Rivières

Abstract

Aspect-Oriented Programming is an emerging software engineering paradigm. It offers new constructs and tools improving separation of crosscutting concerns into single units called aspects. AspectJ, the most used aspect-oriented programming language, represents an extension of Java. In fact, 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 system. However, new dependencies between aspects and classes result in new testing challenges. Interactions between aspects and classes are new sources for program faults. Moreover, existing object-oriented testing methods (unit and integration testing) are not well adapted to the aspect technology. As a consequence, new testing techniques must be developed for aspect-oriented programs. We present, in this paper, a new aspects-classes integration testing strategy and the associated tool. The adopted approach consists of two main phases: (1) static analysis: generating testing sequences based on dynamic interactions between aspects and classes, (2) dynamic analysis: verifying the execution of the selected sequences. We focus, in particular, on the integration of one or more aspects in the control of collaborating classes.

1 INTRODUCTION

Existing object-oriented programming languages suffer from a serious limitation in modularizing adequately crosscutting concerns in a program. Aspect-Oriented Software Development (AOSD) [Aosd05] introduces new abstractions to software engineering dealing with separation of concerns in software development. There have been many approaches to Aspect-Oriented Design (AOD). Each approach attempts to capture and address a significant issue relating to crosscutting in design [Jack05]. Aspect-Oriented Programming (AOP) does for crosscutting concerns what Object-Oriented Programming (OOP) has done for object encapsulation and inheritance: it provides language mechanisms that explicitly capture crosscutting structure and achieve the usual benefits of improved modularity [Kicz01]. The code corresponding to crosscutting concerns is separated into modular units called aspects [Ajpg02]. This reduces the dispersion of the code and tends to improve programs modularity [Mort04, Walk99, Xiet05] making programs easy to maintain, reuse and evolve [Zhao04]. In spite of the many claimed benefits that the aspect paradigm seams offering, it remains that it is not yet mature. AOP introduces, in fact, new dimensions in terms of control and complexity to software engineering and generates new types of faults [Alex04]. Moreover, aspects' features are not covered by existing testing techniques as mentioned by several authors [Alex04, Zhao02, Zhou04]. Consequently, testing aspect-oriented programs is a huge challenge.

The testing process is a crucial issue in software development. It represents an essential task to ensure software quality [Beiz90]. Existing object-oriented testing methods (unit and integration testing) are not well adapted to the aspect technology. The code related to aspects as well as the introduced abstractions and constructs are prone to cause new faults as stated in [Alex04, Mort04]. Moreover, aspects are not complete code units and their behavior often depends on the woven context. In aspect-oriented programs, integration is more fine grained and occurs, as stated in [Alex04], with respect to the intra-method control and data flow. As a consequence, new integration testing techniques must be developed to deal with the new dimensions introduced by aspects. The main difficulty comes from the relationship between aspects and classes. A link between an aspect and a class is not identifiable when analyzing classes [Alex04, Mort04, Xiet05, Zhou04]. One of the major forms of dependencies between aspects and classes comes from the specific relationship “caller/called”. Most of object-oriented testing techniques are based on this type of relationship between classes [Ball98]. In an aspect-oriented system, something different occurs since integration rules are defined in aspects. An aspect describes, using various constructs, how this integration will be done. This additional level of abstraction, and its consequences in terms of control and complexity, must be taken in consideration in order to make sure that dependencies between aspects and classes are tested adequately [Zhou04].

We present, in this paper, a new aspects-classes integration testing strategy and the associated tool. The adopted approach consists of two main phases: (1) static analysis: generating testing sequences based on the dynamic interactions between aspects and classes, (2) dynamic analysis: verifying the execution of the selected sequences. We focus, in particular, on the integration of one or more aspects in the control (interactions) of collaborating classes. The proposed approach follows an iterative process. The first main phase of the strategy consists in the generation of the testing sequences corresponding to the various scenarios of the collaboration between the objects including weaved aspects. The interactions between collaborating classes are specified using UML collaboration diagrams. Aspects are integrated to the original sequences (collaboration diagram) in an incremental way. The second main phase of the strategy supports the verification process of the executed sequences. We focused on AspectJ programs. The proposed technique is, however, general and may be adapted to others aspect implementations. The present work represents an extension of a previous work that focused on a general presentation of the testing sequences generation technique [Mass05-1] and the associated verification process [Mass05-2].

The rest of the paper is organized as follows: in section 2, we present a survey of related works. Section 3 gives the basic concepts of AspectJ. The main steps of the proposed strategy are discussed in section 4. Section 5 presents briefly collaboration diagrams. The proposed testing criteria are discussed in section 6. Section 7 presents the testing sequences generation technique and its illustration on a real case study taken on AspectJ web site [Ajws05]. Section 8 presents the verification process that we implemented and the used fault model. Section 9 illustrates the main functionalities of the developed tool. Finally, section 10 gives a general conclusion and some future work directions.

2 RELATED WORK

Alexander et al. discuss in [Alex04] various types of faults that could occur in aspect-oriented programs. They consider the new dimensions introduced by the integration of aspects into an object code. They propose a fault model including six types of potential sources of errors in an aspect-oriented program. This model constitutes, in our believe, an interesting first basis for developing testing strategies and tools for aspect-oriented programs. Ubayashi and Tamai [Ubay02] have proposed a model checking method for verifying whether or not an aspect-oriented program satisfies expected properties. Mortensen et al. present in [Mort04] an approach combining two traditional testing techniques: structural approach (white box coverage) and mutation testing. Aspects are classified according to whether they modify or not the state of a system. This technique mainly consists in discovering faults that are related to the code introduced by advice. The mutation operators are applied to the process that weaves advice to the object code.

Zhou et al. [Zhou04] suggest a unit testing strategy for aspects. Their approach is presented in four phases. The first step consists in testing classes to eliminate errors that are not related to aspects. Each aspect is integrated and tested individually in a second step. All aspects are integrated and tested in an incremental way. Finally, the system is entirety re-tested. This approach is based on the source code of the program under test. Moreover, Xie et al. [Xiet05] proposed a framework to generate automatically a set of unit tests by using the compiled AspectJ bytecode. In the same context, Zhao [Zhao02] proposes an approach based on control flow graphs. Three testing levels are applied to aspects and classes. The intra-module test is used to test an individual module such as an advice, an introduction, or one aspect/class method. The inter-module test consists in testing one public module in relation to the other modules, which it calls (directly or indirectly). The third testing level aims at testing a whole public module of an aspect or a class when it is randomly called. The strategy proposed by Zhao focuses on unit testing of aspect-oriented programs.

Moreover, other approaches focused on generating testing sequences using state diagrams [Badr05, Xud04, Xud05-1]. Such approaches focus on the behavior of a class where an aspect is weaved. Our research is related to the integration of one or more aspects to the behavior of a group of collaborating objects. The collaboration between several objects specifies how the objects interact dynamically in order to realize a particular task. The problem in this context comes from the aspects integration while they can affect the behavior of the collaboration. We must thus ensure that aspects are integrated correctly into the collaboration. When integrated to the control, aspects have the possibility to change the state of a system as stated in [Mort04]. Concerns implemented in aspects have the potential to extend the original behavior of a given collaboration. Xu [Xud05-2] uses a model-based approach to generate test cases based on interactions between aspects and classes. Their models include class diagrams, aspect diagrams and sequence diagrams. Sereni [Sere03] proposed a method for static analysis of aspects based on a syntactic model of pointcut designators using regular expressions.

3 ASPECTJ: BASIC CONCEPTS

AspectJ represents a seamless aspect-oriented extension of Java [Ajws05, Zhao04]. Eclipse (with AJDT) [Ajpg02] 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 [Redd06]. Moreover, aspects have the possibility to make significant changes to the semantics of a core concern, especially with foreign aspects [Mcea05]. An aspect gathers pointcuts and advice to form a regrouping unit [Ajws05, Balt01, Xiet05]. An aspect is similar to a Java or C++ class in the way that it contains attributes and methods [Zhao04]. The essential mechanism provided for composing an aspect with other classes is called *joint point* [Zhao04]. Even more, join points are well-defined points in the execution in a program [Kicz01]. AspectJ makes it possible to define joint points in relationship to a method call or a class constructor [Balt01]. A pointcut is a set of joint points and aims of referring certain values at those joint points [Kicz01]. A pointcut can be built out of other pointcuts with logical operators (and, or, and not) [Masu03]. 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 [Ajpg02. Ajws05].

4 INTEGRATION STRATEGY: AN ITERATIVE APPROACH

The proposed strategy consists in two main phases (figure 1). The first one is related to the generation of the basic testing sequences corresponding to the collaboration between classes. Each generated sequence corresponds to a particular scenario of the collaboration diagram. Those sequences represent the main scenario (happy path) [Larm03] and its various extensions. We use XML to describe collaboration diagrams and aspects. The proposed strategy consists, in a first step, to generate testing sequences corresponding to all scenarios without aspects integration. This will support the testing process of the collaboration. This step represents an adaptation and extension of some testing sequences generation techniques developed for object-oriented systems [Offu99, Badr03]. The main goal of this step is to verify the collaboration (without aspects) for the realization of a given task and to eliminate faults that are not related to aspects. Aspects are integrated in a second step, in an iterative way. This process is based on the testing criteria presented in section 6. We assume that this will reduce the complexity of the testing process. We focus on the impact of aspects integration on the original scenarios of the collaboration. We formally identify the sequences (scenarios) that are affected by aspects integration. Aspects are introduced automatically, in an incremental way, to the original sequences and tested during the verification process.

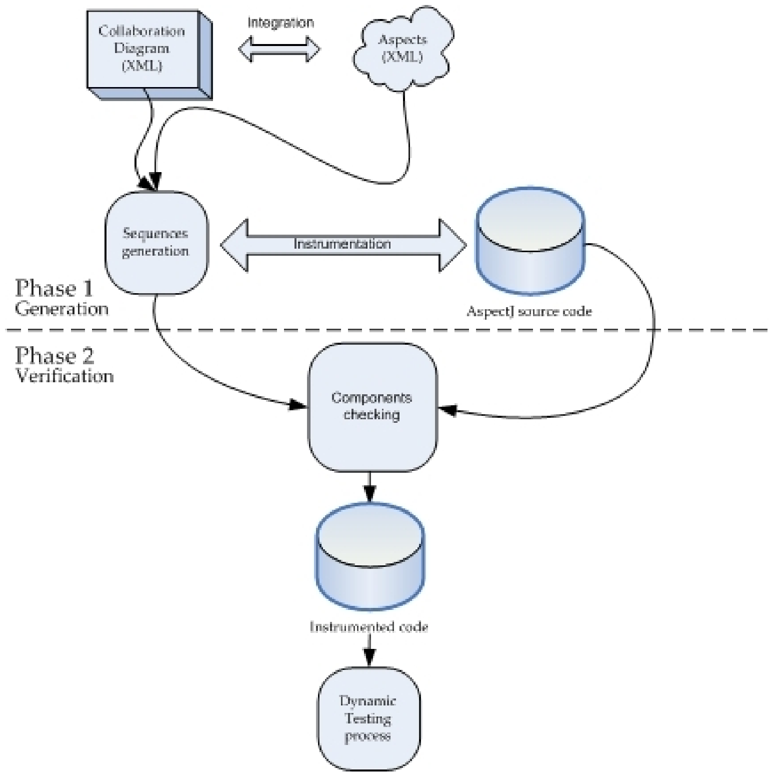


Figure 1. Methodology of the strategy.

The second main phase of the strategy consists in a dynamic analysis which verifies the execution of the implementation of each scenario of the collaboration (including aspects). This process is supported by a source code instrumentation of the program under test. The following algorithm represents the important steps of the strategy.

1. Generating control flow graphs corresponding to the methods implied in the collaboration.
2. Generating messages tree of the collaboration.
3. Generating basic sequences (based on the collaboration between objects).
4. Testing the collaboration between classes based on the various scenarios.
5. Integrating aspects: While there is non integrated aspects
    a. Integrating one aspect.
    b. Identifying sequences that are affected by this integration (following the aspect's control).
    c. Re-testing the affected sequences.
    d. If there are no problems, return to step 5.
6. Testing entirely the collaboration including aspects.
7. End

To instrument the software under test we do use an aspect (generated automatically by our tool) for every sequence under test to capture dynamically all invoked methods in the collaboration (aspects and classes). The instrumented code contains the original source code and an aspect to capture the executed methods. This particular aspect verifies dynamically, among others, if the executed sequence is conform to the selected one (sequence of executed methods, conditions).

5 COLLABORATION DIAGRAMS

In object-oriented systems, objects interact in order to implement behavior. Object-oriented development describes objects and their collaborations [Larm03, Rrsc98]. Behavior can be described at two levels. The first one focuses on the individual behavior of objects while the second one is related to the behavior of a group of collaborating objects [Abdu00]. The collective behavior of a group of collaborating objects, for the realization of a specific task, can be specified using UML collaboration diagrams. UML collaboration diagrams [Wuye02] illustrate the interactions between objects in the form of graphs. Each scenario of the collaboration is represented by a specific sequence. We are interested to the impact of the integration of one or more aspects to a group of collaborating classes. According to the faults model presented by Alexander et al. in [Alex04], two situations could be at the origin of faults. The first one is related to the link that weaves an aspect with its primary abstractions while it introduces new dependencies. The second level is related to the fact that several aspects are integrated to a single class. In that case, it becomes difficult to localize the source of a fault when failures occur. The various control permutation between aspects may complicate the localization of the origin of an error. To reduce this complexity, we adopted an iterative approach for aspects integration. The following criteria aim to cover the new dimensions introduced by the integration of aspects to a group of collaborating classes.

6 TESTING CRITERIA

A testing criterion is a rule or a set of rules that impose conditions on testing strategies [Mort04, Offi99, Xiet05]. It also specifies the required tests in terms of identifiable coverage of the software specification used to evaluate a set of test cases (also known as test suite) [Mort04]. Testing criteria are used to determine what should be tested without telling how to test it. Testing engineers use those criteria to measure the coverage of a test suite in terms of percentage [Offu96]. They are also used to measure the quality of a test suite. The first two criteria are based on collaboration diagrams [Abdu00, Badr04, Offi99, Wuye02]. We extend these criteria to take into account the new dimensions related to the integration of aspects in a collaboration diagram.

Transition coverage criterion

A transition represents an interaction between two objects in a collaboration diagram. Each interaction must be tested at least once [Offu99]. According to Offutt et al. [Offi99], a tester should also test every pre-condition in the specification at least once to ensure that it will always be possible to execute a given scenario (some scenarios might never be executed if a pre-condition is not well-formed). A test will be executed only when the pre-condition related to the transition is true.

C1: Every transition in a collaboration diagram must be tested at least once.

Sequence coverage criterion

The previous criterion relates to testing transitions taken individually. It does not cover transitions sequences [Offu99]. A sequence is a logical suite of several interactions. It represents, in fact, a well-defined scenario in the collaboration that has the possibility to be executed at least once during the program execution. By testing sequences with their control (pre and post condition), we verify all possibilities based on the collaboration diagram (main scenario and its various extensions). In some cases, the number of sequences is unlimited (presence of iterations). The testing engineer has to select the most relevant sequences.

C2: Every valid sequence in a collaboration diagram must be tested at least once.

The first two criteria are related to collaboration diagrams. They do not cover aspects dependencies. Thus, we need to develop new criteria. The following criteria cover the new dimensions introduced by aspects. They are based on the faults model presented by Alexander et al. in [Alex04].

Modified sequences coverage criterion

The collaboration between objects is first tested without the aspects in order to make sure that the various scenarios (interactions between objects) are implemented correctly. An aspect, from its nature, should not modify the semantic of a class [Zhou04]. Aspects depend, in fact, on related classes' context concerning their identity as mentioned by Alexander et al [Alex04]. Therefore, aspects are bounded to classes and they cannot exist by themselves. However, aspects introduce new methods (pieces of code) that must be integrated to the collaboration. Those methods (aspects' methods) can modify the state of the system as stated in [Mort04] and alter, as a consequence, the behavior of a group of collaborating classes. In other terms, aspects will introduce concerns (that must be tested thought) into a group of collaborating classes. It is imperative to adequately test sequences affected by aspects.

C3: Every sequence in the collaboration diagram that is affected by aspects must be re-tested.

Simple integration coverage criterion

Simple integration, as illustrated in figure 2, occurs when only one aspect is integrated to a given class. We need, in this case, to determine formally the affected sequences and test them again.

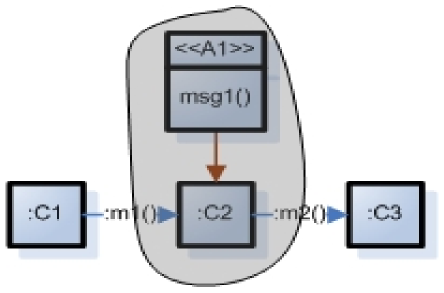


Figure 2. Simple integration.

C4: If a method of a given class is affected by an advice and if that method is used in the collaboration diagram, all sequences that include the execution of that method must be re-tested.

Multi-aspects integration coverage criterion

It is possible that several aspects come to be weaved to a method of a given class (figure 3). In this situation, several 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 a system. Especially, when the aspects are *stateful* or *altering* [Mort04].

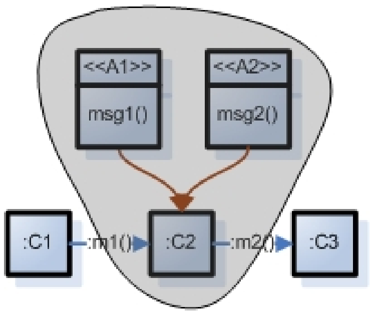


Figure 3. Multi-aspects integration.

C5: If a method of a given class is affected by several advice and if that method is used in the collaboration diagram, all sequences that include the execution of that method must be re-tested. This test will have to be executed with all possible advice permutation after aspects integration.

7 AUTOMATED TESTING SEQUENCES GENERATION

Approach

Testing sequences generation process takes into consideration the control described in the collaboration diagram. Each valid sequence in the collaboration diagram corresponds to a possible scenario that may be executed. The generated sequences also integrate the various interactions between aspects and collaborating classes. The strategy takes into consideration the two levels of integration: classes-classes and aspects-classes integration. It follows an iterative process. The following example (figure 4) has been modeled from a real AspectJ application. This example illustrates some ways that dependent concerns can be encoded with aspects. It uses an example of system comprising a simple model of phone connections to which timing and billing features are added using aspects, where the billing feature depends upon the timing feature. It constitutes, in our opinion, an interesting concrete example to present our approach. We have generated the corresponding collaboration diagram (figure 4) by analyzing the classes implied in the example. For more details about the example see AspectJ web site [Ajws05].

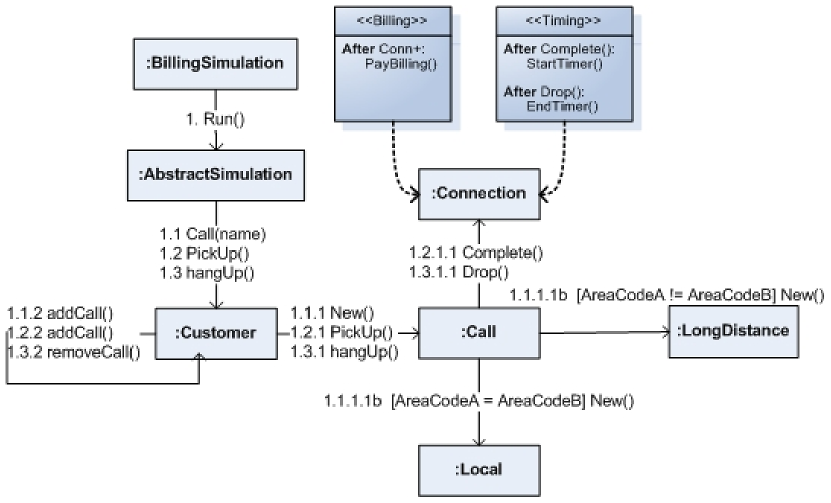


Figure 4. Collaboration diagram.

We start by creating the messages tree corresponding to the collaboration diagram. By analyzing the messages tree, we generate the original sequences. These sequences will allow testing, in a first step, the collaboration between classes (various scenarios). This will allow thereafter, according to the introduced criteria, to determine and visualize the scenarios affected by aspects. Aspects integration is done incrementally. When all aspects are successfully integrated, we test the whole collaboration diagram (including aspects) to ensure that aspects and collaborating classes are working together correctly. This step is also used to determine the possible conflicts, which can be generated by aspects. By integrating incrementally aspects, we expect that the faults related to the interactions between aspects and classes will be relatively easy to identify. The sequences are generated by considering all the possible combinations, for a multi-aspects integration, in order to detect the errors related to the possible random behavior in this case.

Control graphs

Control graphs are used in order to model the control of the methods involved in the collaboration. They are at the basis of the complete control flow graph of the collaboration. It presents a global overview of the control present in the collaboration diagram. Figure 5 shows the control flow graph of the collaboration described in Figure 4.

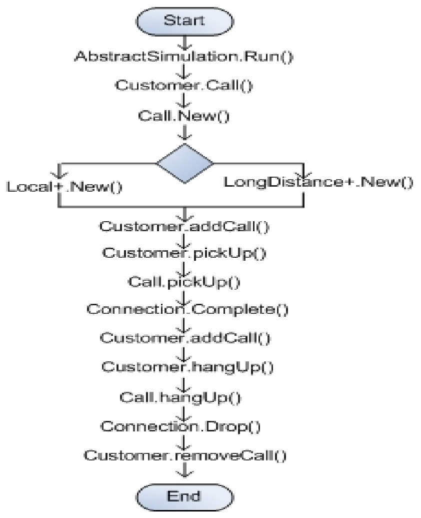


Figure 5. Messages control flow graph.

Messages tree

The control graph related to each method implied in the collaboration is, in fact, translated into a principal sequence (regular expression). The objective at this stage consists in generating the principal sequence of each method. To this end, we use the following notations: The notation {sequence}, expresses zero or several executions of the sequence. The notation (sequence 1 / sequence 2) expresses a mutual exclusion during the execution between sequence 1 and sequence 2. The notation [sequence] expresses that the sequence can be executed or not. Once sequences are created, we use those as a basis for the construction of the main sequence (corresponding to the collaboration) and to generate the corresponding messages tree. Each message is replaced by its own sequence. The substitution process stops when messages are at the leaf levels of the tree. At this step, we do not consider aspects. Figure 6 illustrates the principal sequence of the considered collaboration.

AbstractSimulation.Run(), Customer.Call(),Call.New(),
(Local.New() / LongDistance.New() **)**,
Customer.addCall(), Customer.pickUp(),
Call.pickUp(), Connection.Complete(),
Customer.addCall(), Customer.hangUp(),
Call.hangUp(),Conneciton.Drop(),
Customer.removeCall()

Figure 6. Main sequence.

Main testing sequences

The technique consists of generating, using the messages tree, the control paths starting from the root and taking into account the control (while eliminating the infeasible ones).

Each path will correspond to a particular testing sequence. Every generated sequence represents, in fact, a specific scenario of the collaboration. In order to simplify the notation, we assign a node number to each message in the collaboration. Table 1 shows messages with their assigned node. Table 2 illustrates the generated sequences based on the main sequence presented in figure 6. The initial tests use those sequences to ensure that the collaboration is working correctly.

Node	Classes' Messages
1	AbstractSimulation.Run()
2	Customer.Call()
3	Call.New()
4	Local.New()
5	LongDistance.New()
6	Customer.addCall()
7	Customer.pickUp()
8	Call.pickUp()
9	Connection.Complete()
10	Customer.hangUp()
11	Call.hangUp()
12	Conneciton.Drop()
13	Customer.removeCall()

Table 1. Messages number.

Scenario	Test	Sequences
1	#1	1 → 2 → 3 → 4 → 6 → 7 → 8 → 9 → 6 → 10 → 11 → 12 → 13
2	#2	1 → 2 → 3 → 5→ 6 → 7 → 8 → 9 → 6 → 10 → 11 → 12 → 13

Table 2. Generated base sequences.

Aspects integration

When all basic sequences related to the collaboration between classes are generated and tested, we proceed to aspects integration. Aspects are integrated in an incremental way, as mentioned previously, to facilitate errors detection. We adopted an iterative strategy by starting with the most complex aspect. According to the criteria established in section 6, we determine the sequences to which the aspects are weaved. These sequences will be re-tested including the fragments of code introduced by aspects. The code introduced by aspects will be executed when the corresponding join point will be reached. When all aspects are entirely integrated, we re-test all sequences one more time to ensure that the collaboration, including the aspects, works correctly. According to the collaboration diagram given by figure 4, the aspect *Timing* introduces two methods to the main sequences. Consequently, advice integration starts with this aspect. The proceeding order to introduce advice does not have importance. We begin with *StartTimer* and *EndTimer*. Table 3 shows the methods introduced by the *Timing* aspect with their associated node number. The obtained sequences are presented in table 4.

Node	Aspect's Methods
14	StartTimer ()
15	EndTimer()

Table 3. Methods introduced by the *Timing* aspect.

New method	Scenario	Test	New sequences
14	1	#3	1 → 2 → 3 → 4 → 6 → 7 → 8 → 9 → **14** → 6 → 10 → 11 → 12 → 13
	2	#4	1 → 2 → 3 → 5→ 6 → 7 → 8 → 9 → **14** → 6 → 10 → 11 → 12 → 13
15	1	#5	1 → 2 → 3 → 4 → 6 → 7 → 8 → 9 → 6 → 10 → 11 → 12 → **15** → 13
	2	#6	1 → 2 → 3 → 5→ 6 → 7 → 8 → 9 → 6 → 10 → 11 → 12 → **15** → 13
14 + 15	1	#7	1 → 2 → 3 → 4 → 6 → 7 → 8 → 9 → **14** → 6 → 10 → 11 → 12 → **15** → 13
	2	#8	1 → 2 → 3 → 5→ 6 → 7 → 8 → 9 → **14** → 6 → 10 → 11 → 12 → **15** → 13

Table 4. Integration for the *Timing* aspect.

After that we integrate the aspect *Billing*. This aspect is particular because its pointcut points on a class constructor. The main problem comes from the fact that this class is an abstract class. The advice related to that pointcut will be triggered when one of the sub-classes will be instantiated. The implementation for the connection class is done in two sub-classes: *Local* and *Longdistance*. Thus, every instance of those two classes will trigger the pointcut defined in the aspect *Billing*. Table 5 presents the method introduced by the *Billing* aspect. The new sequences are shown in the table 6.

Node	Aspect's Method
16	PayBilling()

Table 5. Methods introduced by the *Billing* aspect.

New method	Scenario	Test	New sequences
16	1	#9	1 → 2 → 3 → 4 → **16**→ 6 → 7 → 8 → 9 → 6 → 10 → 11 → 12 → 13
	2	#10	1 → 2 → 3 → 5 → **16**→ 6 → 7 → 8 → 9 → 6 → 10 → 11 → 12 → 13

Table 6. Integration for the *Billing* aspect.

Once the integration of all aspects is done, we test the system entirely by integrating all aspects to the collaboration diagram. Knowing that we have two scenarios, the last two tests need be applied (table 7).

Scenario	Test	New sequences
1	#11	1 → 2 → 3 → 4 → **16** → 6 → 7 → 8 → 9 → **14** → 6 → 10 → 11 → 12 → **15** → 13
2	#12	1 → 2 → 3 → 5→ **16** → 6 → 7 → 8 → 9 → **14** → 6 → 10 → 11 → 12 → **15** → 13

Table 7. Global integration.

8 TESTING PROCESS

The testing process aims essentially to verify if the executed sequences are in accordance with the selected ones in one hand, and if the obtained results are conform to the expected ones in the other hand. We present in what follows the main phases of the testing process. For each generated sequence S_i:

1. Instrumenting the program under test
2. Executing the program under test
3. Analyzing the results

Instrumenting the program under test

When all sequences are generated, we can start the testing process. In opposition to traditional instrumentation techniques, we do use aspects to capture dynamically a trace of the executed methods in a given sequence. The advantage of our approach is that we don't modify in any way the original source code of the program we are testing. Traditional instrumentation techniques consist generally in introducing many lines of source code in the program under test. Those fragments of code may introduce involuntary faults [Beiz90]. We generate an aspect for each sequence under test. When we want to test a specific sequence, we compile the program with the corresponding aspect. The tracking aspects are automatically generated by our tool and are functional with any AspectJ [Ajws05] program. In fact, our strategy is general and would be easily adaptable to another aspect implementation. When a method involved in a sequence is executed, the tracking aspect will keep information about that execution. This information will be used in the following steps (verification process, testing coverage).

Executing the program under test

When the instrumentation phase is completed, we can execute the program. It mainly consists to run the program and test a specific sequence. It remains to the tester to provide testing data to ensure the execution of the selected sequence.

Analyzing results

When a sequence has been successfully executed, an analyzer compares the executed methods with the expected ones. Our strategy essentially consists to discover three types of faults.

1. Specifications based faults.
2. Pre-condition based faults.
3. Source code based faults (java exceptions).

Specification based faults

This kind of fault occurs when a part of the specification is not well transposed to the implementation. A missing method, an invalid method signature could be source of this type of fault. This level mainly consists to verify if an executed sequence is in accordance with the expected one.

Pre-condition based faults

It is possible to insert pre-conditions in a collaboration diagram. In fact, a pre-condition may be attached to a method or an advice. We must test all pre-conditions in the collaboration diagram in order to verify if all scenarios can be executed. If a pre-condition is always *false* some sequences might be never executed. This fault level aims essentially to test all pre-conditions in the collaboration diagram.

Source code based faults (java exceptions)

One of the best features about java is its capacity to handle errors when they occur. Furthermore, it is possible with AspectJ [Ajws05] to collect those errors. While we automatically create aspects to track the executed methods we also generate pointcuts that will catch all exceptions thrown by java. The source code fault level is capable to capture every kind of exception thrown by Java (SAXException, InterruptedException, IOException, InterruptedException …) since they are based on the *Exception* class.

9 A TOOL SUPPORTING OUR STRATEGY

We developed a tool supporting our strategy. It supports the testing sequences generation as well as the verification process. We used the AspectJ technology. Specially, the Eclipse framework [Ajws05] was the main architecture we used. Figure 7 presents the architecture of the tool and figure 8 its main interface. The tool will be illustrated, in what follows, using the case study presented in section 7.

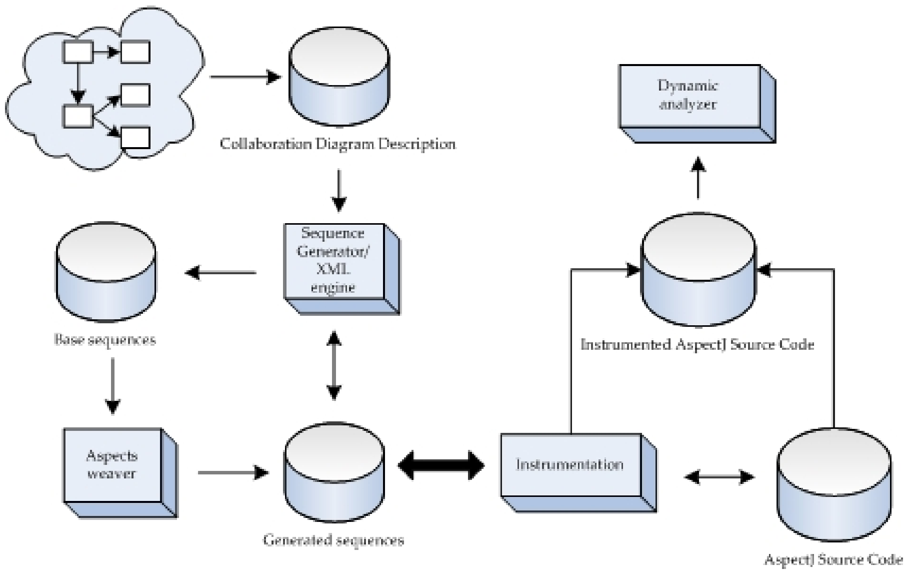
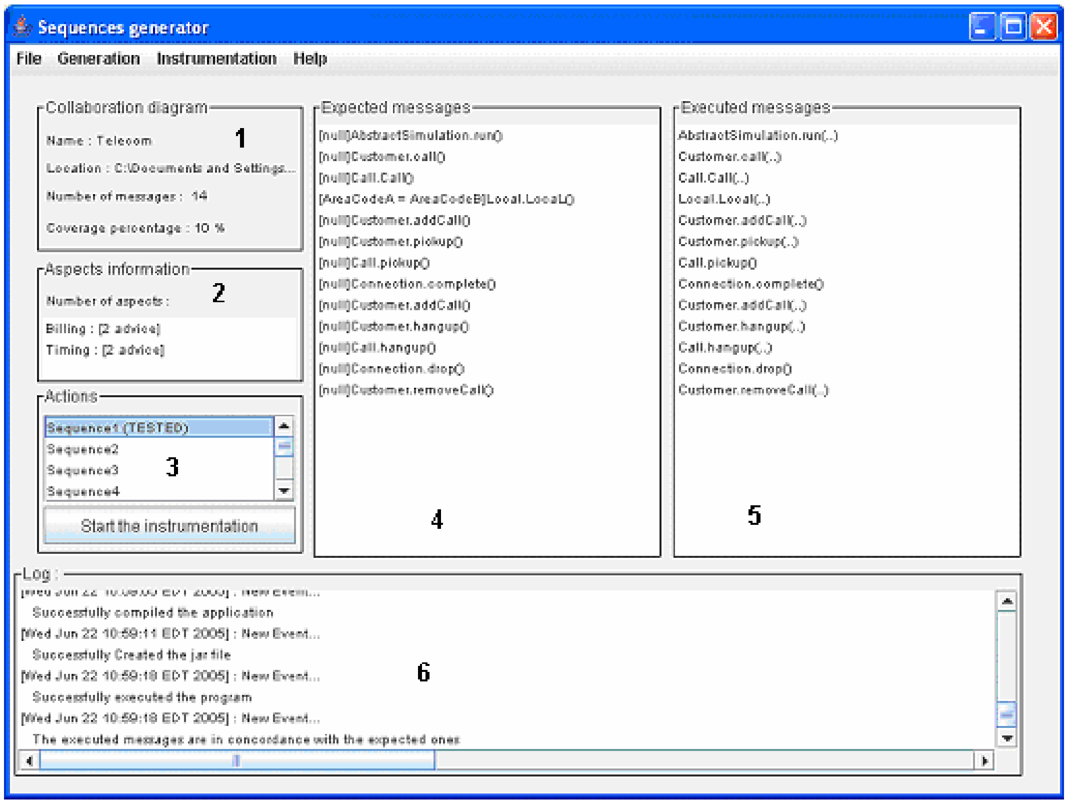


Figure 7. Architecture of the tool.



Name : Telecom

Location : C:\Documents and Settings...

Number of messages : 14

Coverage percentage : 10 %

Number of aspects :

Billing : [2 advice]

Timing : [2 advice]

Sequence1 (TESTED)
Sequence2
Sequence3
Sequence4

```
[null]AbstractSimulation.run()
[null]Customer.call()
[null]Call.Call()
[AreaCodeA = AreaCodeB]Local.Local()
[null]Customer.addCall()
[null]Customer.pickup()
[null]Call.pickup()
[null]Connection.complete()
[null]Customer.addCall()
[null]Customer.hangup()
[null]Call.hangup()
[null]Connection.drop()
[null]Customer.removeCall()
```

```
AbstractSimulation.run(..)
Customer.call(..)
Call.Call(..)
Local.Local(..)
Customer.addCall(..)
Customer.pickup(..)
Call.pickup()
Connection.complete()
Customer.addCall(..)
Customer.hangup(..)
Call.hangup(..)
Connection.drop()
Customer.removeCall(..)
```

```
[Wed Jun 22 10:59:05 EDT 2005] : New Event...
  Successfully compiled the application
[Wed Jun 22 10:59:11 EDT 2005] : New Event...
  Successfully Created the jar file
[Wed Jun 22 10:59:18 EDT 2005] : New Event...
  Successfully executed the program
[Wed Jun 22 10:59:18 EDT 2005] : New Event...
  The executed messages are in concordance with the expected ones
```

Figure 8. Main interface of our tool.

Generating testing sequences

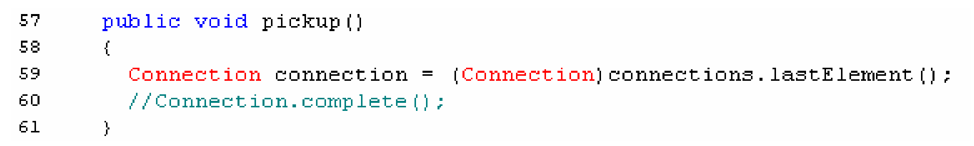
The generating process is based on the criteria presented in section 6. It is based on collaboration diagrams and related aspects. As mentioned in section 4, collaboration diagrams and related aspects are described using XML. When we select a sequence, all involved methods appear in the *Expected messages* frame 4 (figure 8) with their pre-condition. In our example, as illustrated in section 7, we will have twelve sequences. Those sequences will be used in the dynamic testing process.

Dynamic testing process

To test the generated sequences, we execute the program under test and capture the executed methods. For the three levels of faults represented in our fault model, we introduced an error in the program. It was mainly to illustrate how our tool identifies the source and the type of each error. We show, in what follows, the introduced errors in the Java source code of the program under test and how the errors was detected by our tool.

Specification based faults

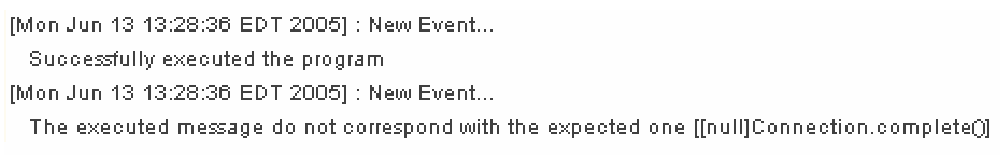
To produce a specification-based fault, we intentionally omitted to execute a method. In figure 9, the line 60 has been placed as a comment. In that case, the method *complete* defined in the class *Connection* won't be called. If we examine the generated sequences, we can see that the *complete* method is involved in all sequences (see section 7). Thus, every tested sequence should throw an error.



```
57      public void pickup()
58      {
59        Connection connection = (Connection)connections.lastElement();
60        //Connection.complete();
61      }
```

Figure 9. A specification based fault.

Let us consider the first sequence. When we execute the program under test the following message (figure 10) appears to the tester in window 6 of the main interface of our tool.



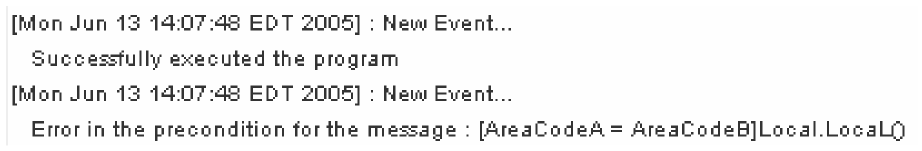
```
[Mon Jun 13 13:28:36 EDT 2005] : New Event...
  Successfully executed the program
[Mon Jun 13 13:28:36 EDT 2005] : New Event...
  The executed message do not correspond with the expected one [[null]Connection.complete()]
```

Figure 10. Error related to a specification based fault.

The tool informs the tester that there's a problem with the *complete* method and the executed sequence do not conform to the expected one.

Pre-condition based faults

As illustrated in section 7, we have two base sequences which represent the two scenarios described in the collaboration diagram. The pre-condition related to the *New* method in *Local* and *LongDistance* classes determine the executed sequence. To verify if our tool could handle a pre-condition fault, we provided input data that will execute the second scenario (long distance call). The program under test will simulate a long distance call while our analyzer expects a local call. A pre-condition fault will be thrown by our analyzer (figure 11) and appears in window 6 (figure 8).



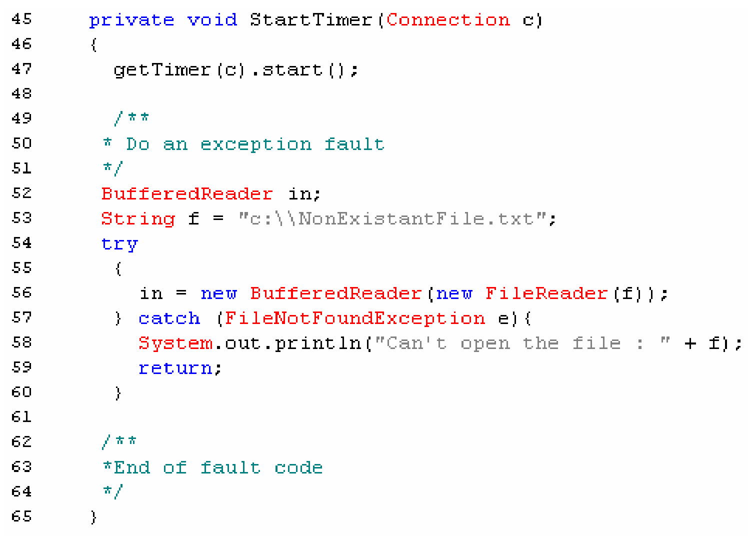
```
[Mon Jun 13 14:07:48 EDT 2005] : New Event...
  Successfully executed the program
[Mon Jun 13 14:07:48 EDT 2005] : New Event...
  Error in the precondition for the message : [AreaCodeA = AreaCodeB]Local.LocaL()
```

Figure 11. Error about a pre-condition based fault.

The error shows that there’s a problem with the corresponding pre-condition. Moreover, the precondition *AreaCodeA* = *AreaCodeB* was false while we were expecting it true at the execution.

Exception based faults

In our example (*Telecom*), we inserted some code that will produce an IOException in the Timing aspect.



```
private void StartTimer(Connection c)
{
  getTimer(c).start();

  /**
 * Do an exception fault
 */
 BufferedReader in;
 String f = "c:\\NonExistantFile.txt";
 try
  {
    in = new BufferedReader(new FileReader(f));
  } catch (FileNotFoundException e){
    System.out.println("Can't open the file : " + f);
    return;
  }

 /**
 *End of fault code
 */
}
```

Figure 12. An exception based fault.

The introduced code (lines 49 to 64) in figure 12 aims to open a non existing file (*NonExistantFile.txt*). Since the file does not exist, java will throw an IO exception. The tracking aspect will find the error and a message will inform the tester where exactly the fault has been detected.

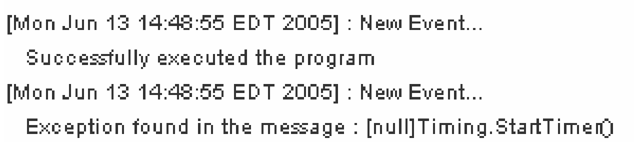

```
[Mon Jun 13 14:48:55 EDT 2005] : New Event...
   Successfully executed the program
[Mon Jun 13 14:48:55 EDT 2005] : New Event...
   Exception found in the message : [null]Timing.StartTimer()
```

Figure 13. Error about an exception based fault.

The message (figure 13) informs that the fault has been located in the *StartTimer* method in the aspect *Timing*. In that case, it remains to the tester to check the method and find exactly the source of the error.

Testing coverage

The implemented approach allows computing the testing coverage according to the tested sequences. This information (in percentage) is given in frame 1 (figure 8). We have defined, in fact, two types of testing coverage. The first one is in relation with the simple integration criterion while the second is in relation with the multi-aspects integration criterion. In each case, we compute the testing coverage as:

$$SC = \frac{\textit{Number of Executed Sequences (NES)}}{\textit{Number of Generated Sequences (NGS)}}$$

10 CONCLUSION AND FUTURE WORK

We presented, in this paper, a new integration testing strategy for aspect-oriented programs and the associated tool. We focused on the integration of one or more aspects to the control of collaborating classes. Our methodology is based on UML collaboration diagrams. It offers, compared to the code based approaches, the advantage of preparing the testing process early in the software development.

The adopted approach consists of two main phases: (1) static analysis: generating testing sequences based on dynamic interactions between aspects and classes. (2) dynamic analysis: verifying the execution of the selected sequences. We use XML schemas to describe collaboration diagrams and related aspects. The strategy follows an iterative process. It is supported by a tool. We focused on AspectJ programs. The proposed technique is, however, general and may be adapted to other aspect implementations. As future work, we plan to integrate our tool to the eclipse platform (plug-in) and to experiment it on real AspectJ programs.

ACKNOWLEDGMENTS

This research was supported by NSERC (Natural Sciences and Engineering Research Council of Canada) grants.

REFERENCES

[Abdu00] A. Abdurazik and J. Offutt, Using UML Collaboration Diagrams for Static Checking and Test Generation, In Third Internationl Conference on The Unified Modeling Language (UML '00), York, UK, October 2000.

[Alex04] R. Alexander, J. Bieman and A. Andrews, Towards the Systematic Testing of Aspect-Oriented Programs, Technical Report CS-4-105, Colorado State University, Fort Collins, Colorado, USA, March 2004.

[Aosd05] Aspect-Oriented Software Development Web Site (AOSD), http://.aosd.net/

[Ajpg02] T. AspectJ, The AspectJ™ Programming Guide, 2002.

[Ajws05] AspectJ Web Site, http://eclipse.org/aspectj/

[Badr05] M. Badri, L. Badri and M. Bourque-Fortin, Generating Unit Test Sequences for Aspect-Oriented Programs, 3rd International Conference on Information and Communication Technology (ICICT'2005), IEEE Computer Society Press, Cairo, Egypt, December 2005.

[Badr04] M. Badri, L. Badri, and M. Naha, A use Case Driven Testing Process: Toward a Formal Approach Based on UML Collaboration Diagrams, Post-Proceedings of FATES (Formal Approaches to Testing of Software) 2003, in LNCS 2931, Springer-Verlag, January 2004.

[Ball98] 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), volume 23,2 of ACM Software Engineering Notes, New York, March 1998.

[Balt01] J. Baltus, La Programmation Orientée Aspect et AspectJ : Présentation et Application dans un Système Distribué, Mini-Workshop: Systèmes Coopératifs. Matière Approfondie, Institut d'informatique, Namur, 2001.

[Beiz90] B. Beizer, Software Testing Techniques, International Thomson Comuter Press, 1990.

[Jack05] A. Jackson and S. Clarke, Towards a Generic Aspect Oriented Design Process, 7th International Workshop on Aspect-Oriented Modeling, (AOM 2005) Models 2005, Jamaica.

[Kicz01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. Griswold, An Overview of AspectJ, In Lecture Notes in Computer Science, Volume 2072, p. 327-355, 2001.

[Larm03] C. Larman, UML et les Design Patterns,2e édition, CampusPress 2003.

[Mass05-1] P. Massicotte, M. Badri and L. Badri, Generating Aspects-Classes Integration Testing Sequences: A Collaboration Diagram Based Strategy. 3rd ACIS International Conference on Software Engineering Research, Management & Applications (SERA2005), IEEE Computer Society Press, Central Michigan University, Mt. Pleasant, Michigan, USA, August 2005.

[Mass05-2] P. Massicotte, M. Badri and L. Badri, Aspects-Classes Integration Testing Strategy: An incremental approach. 2nd International Workshop on Rapid Integration of Software Engineering techniques (RISE 2005), Lectures Notes in Computer Science, Heraklion, Crete, GREECE, September 2005.

[Masu03] H. Masuhara and G. Kiczales, Modeling Crosscutting in Aspect-Oriented Mechanisms, In Proceedings of ECOOP 2003, LNCS #2743, pp.2-28, 2003.

[Mcea05] 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.

[Mort04] 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.

[Offu99] J. Offutt and A. Abdurazik, Generating Tests from UML Specications, In Second International Conference on the Unified Modeling Language (UML ’99), Fort Collins, CO, October 1999.

[Offu96] J. Offutt and J. Voas, Subsumption of Condition Coverage Techniques by Mutation Testing, ISSE-TR-96-01, January 1996.

[Offi99] J. Offutt, Y. Xiong and S. Liu, Criteria for Generating Specification-based Tests, In Engineering of Complex Computer Systems, ICECCS '99, 1999.

[Redd06] 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.

[Rrsc98] Rational Software Corporation. Rational Rose 98: Using Rational Rose, Rational Rose Corporation, Cupertino CA, 1998.

[Sere03] D. Sereni and O. de Moor. Static analysis of aspects. In Proceedings of the 2nd International Conference on Aspect-Oriented Software Development, March 2003.

[Ubay02] N. Ubayashi and T. Tamai. Aspect-oriented programming with model checking. In Proceedings of the 1st International Conference on Aspect-oriented software development, April 2002.

[Walk99] 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.

[Wuye02] Y. 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.

[Xiet05] T. Xie, J. Zhao, 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.

[Xud04] D. Xu, W. Xu and K. Nygard, A State-Based Approach to Testing Aspect-Oriented Programs, Technical report, North Dakota University, Department of Computer Science, USA, 2004.

[Xud05-1] D. Xu, Test Generation from Aspect-Oriented State Models. Technical Report, NDSU-CS-TR-05-XU02, Sept 2005.

[Xud05-2] D. Xu and W. Xu, A Model-Based Approach to Test Generation for Aspect-Oriented Programs, AOSD'05 Workshop on Testing Aspect-Oriented Programs. Chicago, March 2005.

[Zhao02] J. Zhao, Tool support for unit testing of aspect-oriented software, In Proceedings OOPSLA' 2002 Workshop on Tools for Aspect-Oriented Software Development, November 2002.

[Zhao04] 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.

[Zhou04] Y. Zhou, D. Richardson, and H. Ziv, Towards a Practical Approach to test aspect-oriented software, In Proc. 2004 Workshop on Testing Component-based Systems (TECOS 2004), Net.ObjectiveDays, September 2004.

About the authors

Philippe Massicotte (philippe.massicotte@uqtr.ca) has recently received his master in computer science from the University of Quebec at Trois-Rivières and will start a PhD in computer science in January 2007. His main areas of interest include aspect-oriented programming, software quality, formal methods as well as various topics of software engineering.

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.

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.