Is AOP code easier or harder to test than OOP code?

Mariano Ceccato, Paolo Tonella and Filippo Ricca
ITC-irst
Centro per la Ricerca Scientifica e Tecnologica
38050 Povo (Trento), Italy
{ceccato, tonella, ricca}@itc.it

Abstract

The adoption of traditional testing techniques with AOP systems is expected to be quite hard, because of the novel constructs offered by AOP. For example, testing should validate the pointcut designators, which define the execution points at which aspects apply. These may be difficult to test when they involve complex dynamic conditions, depending on the execution stack. Other sources of difficulties are associated with the aspect composition order, with the inter-type declarations, and with the changes in normal and exceptional control flow, possibly introduced by aspects.

On the other hand, a novel, aspect oriented approach to testing could be devised, which takes advantage of the separation of concerns implemented in AOP code, in order to extend the benefits of such separation to the testing phase. In this paper, an incremental testing process is considered, which allows testing the base code and the crosscutting functionalities, implemented as aspects, in separate, successive steps.

1 Introduction

Aspect Oriented Programming (AOP) supports the separation of the crosscutting concerns by means of pointcuts, which intercept the execution at given join points, and introductions, which add or alter features of the base code. Such constructs have a large impact on the dynamic behavior of the affected program, so that testing is expected to become harder.

Actually, AOP requires that the execution points to be intercepted by an aspect are specified declaratively in the aspect itself. This might be an error prone task that causes faults. Moreover, redirecting the execution to another code fragment (contained in an aspect) might break the contract between caller and callee in a method execution or might alter the state of the current object in an incorrect way. Changes in the execution order of the statements might also violate some assumptions required for a given statement to work correctly. The exceptional execution flow can be altered by an aspect as well, possibly in an undesired way. Finally, static crosscutting (introductions) might make invalid assumptions on object types and might change the polymorphic calls. All these characteristics of AOP make testing challenging and demand for novel approaches, especially when the existing ones cannot be adapted to reveal the faults associated with some of the new constructs.

On the other hand, AOP simplifies design and coding by separating the principal decomposition from the crosscutting concerns. If during the testing phase such a separation is maintained, we can expect that the benefits of AOP can be extended to testing as well. This requires the definition of a novel testing process, in which the base code is tested separately from the crosscutting concerns, which are added incrementally. Moreover, during the successive integration of new aspects, re-test of the base application can be limited to those code portions possibly affected by the aspects, thus achieving a separation of the test of the aspects.

In this paper, we discuss why AOP testing is expected to be harder (Section 2) and why easier (Section 4) than testing more traditional Object Oriented Programming (OOP) code. In Section 3 we consider AOP-specific testing techniques that could mitigate the novel difficulties of AOP testing. Overall, the authors' position is that the benefits of aspect orientation *can* be extended to the testing phase, by adopting a proper testing process and using proper testing techniques. A description of the related works (Section 5) and of our future works (Section 6) conclude the paper.

2 Why is it harder?

Some of the faults that are generated while developing AOP code are quite new, with respect to the traditional case. An attempt to understand what are the special features of the faults introduced by the new paradigm was made by

Alexander et al. [**?**], who defined a fault model, providing a categorization of the AOP specific faults. In the following, the fault model by Alexander et al. is briefly summarized and slightly extended with a couple of additional fault types. It will constitute the basis for the further discussion presented in this paper.

2.1 Fault model

1. **Incorrect strength in pointcut patterns:** The patterns used in pointcut designators give the set of the join points to be intercepted. A fault might occur when the pointcut designator strength is not accurate and a wrong join point set is intercepted. This involves two cases:

   - When the designator is too weak, the accepted join point set is not restrictive enough. The pointcut intercepts more join points than the ones required by the crosscutting concern.
   - Conversely, when the designator is too strong, the set is restricted excessively and the pointcut intercepts less points than required. Therefore, the concerns partially fails to apply in cases where it should.

2. **Incorrect aspect precedence:** When the same code portion is affected by more than one aspect, depending on the order in which aspects are woven to the base code, differences can occur. When no composition precedence is defined, all the possible compositions are potential instances to be considered.

3. **Failure to establish expected postconditions:** Postconditions that were valid for the base code, before adding the crosscutting concerns, represent the contract between calling and called methods. These contracts should keep valid also when aspects are weaved. When advices produce ripple effects that violate some postconditions, the overall system behavior might be incorrect.

4. **Failure to preserve state invariants:** Similarly to the postconditions, state invariants should keep valid also for the weaved application. Aspects violating them might corrupt the behavior of the overall system.

5. **Incorrect focus of control flow:** Pointcut designators that contain conditions on the execution stack (e.g., using the `cflow` primitive pointcut) define a join point set that cannot be evaluated statically. Therefore, errors can be hidden in them which are difficult to expose, in that they require very specific execution conditions to hold.

6. **Incorrect changes in control dependencies:** Around-advices replace the original execution with a new one, defined in the aspect. In the advice body, the original execution can be resumed at any point, thus changing the original control flow structure that determines its execution. Thus, defects may descend from assumptions on the control dependency structure that become invalid in the aspect code.

7. **Incorrect changes in exceptional control flow:** Advices containing statements that possibly throw an exception might cause an implicit modification in the system control flow, because such an exception triggers the execution of an appropriate `catch` statement, either in the aspect code or down in the base code. Moreover, when the `declare soft` construct is used to modify the system exception handling mechanism, different branches might be taken in the original and in the aspectized code, since an exception originally raised during the execution is now softened.

8. **Failures due to inter-type declarations:** Static crosscutting modifies the base code by means of introductions and intertype declarations. This mechanisms could produce ripple effects in the control flow, each time the control flow depends on the static class structure. For example, the boolean operation `x instanceof Y` produces a *true* value whenever the dynamic type of the object `x` is a subtype of `Y`. If the inheritance relation of the type of `x` is changed by an aspect, the previous boolean operator could return a different value, causing a different branch to be taken, thus, altering the control flow.

9. **Incorrect changes in polymorphic calls:** Modifications in the system behavior may occur when a method introduction is used to override a method inherited from a super class. In fact, before weaving the aspect, any invocation to such a method was redirected to the method in the super class, while, after weaving, the same invocation is dispatched to the introduction.

2.2 Adaptation of existing techniques

In all the join points selected by a pointcut designator, there is an implicit branch that transfers the control to the associated advice. Thus, an extension of coverage testing and more specifically of the branch coverage criterion to AOP seems a reasonable choice in order to expose faults related to an incorrect strength in pointcut patterns (type 1). In case of a too weak pointcut designator (too large join point set), test cases traversing the incorrectly added branches, departing from incorrectly intercepted join points, might expose this type of fault. Conversely, when branches are missing from execution points to advices, traditional test cases

should expose the fault whenever the observable behavior is altered by the missed advice execution.

Faults related to concerns that violate postconditions and invariants (type 3 and 4) of the base code can be exposed by test cases whose observable output depends on such postconditions and invariants. Adaptation of data-flow criteria might help here, since such violations are usually associated with changes in the definition-use pairs.

An adapted version of the branch coverage criterion, that treats the advices as regular method invocations with parameters, seems to be a reasonable approach to expose faults coming from static crosscutting (type 8 and 9) and associated with a changed normal/exceptional control flow (type 6 and 7). In fact, an incorrect branch is taken in all these cases. Thus, exercising each branch in at least one test case ensures that every incorrect branch is executed at least once during testing.

The only two fault types that seem to be not adequately tested by means of extensions of traditional techniques are type 2 and 5. They demand for AOP specific techniques. Our proposal for two such techniques is given in Section 3.

2.3 Tangling in test cases

The adoption of traditional testing methods for AOP systems tends to produce complex test cases.

Traditional coverage criteria can be based on the actual control flow that the system has, once woven. This requires to resolve pointcuts and to weave advices and introductions into the base code. The resulting control flow corresponds to the base system behavior tangled with all the crosscutting concerns. Thus, the same test case exercises both the principal decomposition and all the crosscutting concerns at the same time. Consequently, the definition of the test inputs and of the expected outputs (oracle) for the test cases requires to reason about the base application and all the tangled concerns as a whole. In other words, the advantages of the separation of concerns that were achieved in the previous development phases are lost when coming to testing, if the only way to test the application consists of testing the weaved code. A novel approach to untangle the test cases for the crosscutting concerns from those for the base code seems to be necessary. This is discussed in detail in Section 4.

3 AOP specific testing techniques

Specific testing techniques are necessary to expose the faults of type 2 and 5 in the fault model described in Section 2. These are the two fault types for which existing techniques are clearly not adequate. However, even if in principle the other fault types can be exposed using existing techniques, properly adapted, it might be the case that more effective techniques to expose them do exist. Thus, we are not claiming that the other fault types do not deserve investigation of specific techniques. We are just considering that existing techniques are – to some extent – adequate to expose them.

3.1 Designator Coverage

The testing criterion we propose in this sub-section, called *designator coverage*, aims at exposing faults if type 5. A pointcut designator can not be fully resolved at weave time if it contains primitive dynamic pointcuts, such as `cflow` and `within`. In fact, these primitive pointcuts require to evaluate conditions on the run-time execution stack.

Given a pointcut designator with dynamic pointcuts, the number of the execution stacks that satisfy it is, in general, unbounded. So, the exhaustive test of all those satisfying a pointcut designator is not feasible. As suggested by Alexander et al. [**?**], the case is similar to the test of the conditions in conditional or loop statements, with the remarkable difference that the number of combinations to consider is not defined upon all the possible boolean values (a finite set), but is defined upon all the possible execution stacks.

Our proposal was inspired by the path coverage criteria and the way they deal with the possibly unbounded number of loop traversals. One of the most widely adopted solution is to use k-limiting for the loops, i.e., all paths must be covered such that loops are traversed 0 to k times, with k a small constant, often set to 2.

Given the static set J of the join points corresponding to a pointcut designator and obtained by replacing the dynamic conditions with *true*, let us consider all the combinations of execution stacks that are obtained by pushing each join point $j \in J$ onto the stack 0 to k times. Several of these execution stacks are clearly infeasible, being not compatible with (a static approximation of) the call graph of the given application. All the remaining execution stacks must be exercised in at least one test case in order for our *designator coverage* criterion to be satisfied. Similarly to the infeasible path problem, it might happen that some of the execution stacks required by this criterion are infeasible (it is easy to show that their detection remains in general undecidable). Thus, we cannot expect to reach 100% desigator coverage for any given AOP program.

3.2 Composition Coverage

When the weaving order of the aspects makes a difference, dominance constraints should be used to specify the correct precedences. In absence of such constraints, composition errors might occur. A conservative solution to detect them, proposed in [**?**], consists of requiring that all the possible weaving configurations are tested.

An improvement of this method consists of testing only those configurations that differ in at least one data dependency. We can thus define a *composition coverage* criterion, requiring that a composition order is tested only if it changes at least one data dependency with respect to any of the previously tested composition orders.

4 Why is it easier?

If separate concerns can be tested separately, testing an AOP application becomes easier than testing a traditional application, where the crosscutting concerns are tangled with the base code. In order to test the aspects separately from the base code, we must address the following two problems:

1. How do we test the base code in isolation?
2. How do we determine the code to be re-tested when aspects are added?

The incremental testing method described below tries to address both issues.

4.1 Incremental AOP testing

Let us consider, for simplicity, the branch coverage criterion. Incremental AOP testing starts by applying branch coverage to the base code, without considering the aspects. The resulting test suite provides a way to expose those errors that affect the principal decomposition. After that, the method goes on considering the first crosscutting concern. It is woven into the base code and the coverage for the previous suite is evaluated on this composition. The existing test suite is expected to be lacking in covering the composition, because new code has been introduced and because aspects have possibly modified the base control structure and behavior. Therefore, a second collection of test cases must be defined, in order to reach an adequate coverage of this second version of the application. Incremental AOP testing continues in this way, until all aspects are weaved and the final, complete application is tested.

The first problem we must consider is how to test a partially woven application, i.e., an application that misses the implementation of some of its crosscutting functionalities. The solution commonly adopted in integration testing consists of defining *stubs* and *drivers* that simulate the missing parts. For example, a persistence concern that mediates the communication between the main application and the data base can be required by the principal decomposition, in that the application relies on the concern to obtain all the references to the object instances. The persistence concern can be replaced by a stub, which returns objects from a fixed pool, instead of accessing the data base. In such a way the application can be run and tested separately from the persistence aspect.

The second problem we face is how to determine which test cases must be re-run when an aspect is added to the (partially woven) application. This problem is well known in the testing field and is referred to as the *test selection* problem. It is usually considered in the context of regression testing [**?**, 1]. Available methods are based on a syntactic or semantic computation of the differences between the two versions of the given application. Only those test cases that are modification-traversing are selected for re-execution. Such selection techniques are *safe* whenever they ensure that (under controlled testing) no fault can ever be missed. In other words, it can be shown that the execution of the discarded test cases would give the same output in both versions of the application.

The incremental approach may lead to a more effective debugging, because it separates the bugs in the base code from those in the aspect code or deriving from the interaction between base code and aspects. The test suites are, in fact, divided according to the concerns defined during the design and development phases. Thus, tests should be easier to understand, being better modularized.

We considered incremental AOP testing in the context of coverage (white-box) testing. However, it can be used also with a black-box approach, in that it is focused on splitting the test cases according to the system functionalities, which might belong to the principal decomposition or might be crosscutting concerns.

5 Related Works

In [**?**], a fault model is introduced to categorize the typical faults encountered while developing AOP software. Faults may be associated with an incorrect strength of the pointcut designators. Composition faults can arise when more than one weaving order exists among aspects, but no order constraint is defined. Other faults may arise when negating those invariants or postconditions that are valid for the base code, or when modifying the control dependencies in an unexpected way. Testing criteria are proposed in order to detect these new faults, mainly using adaptation of existing testing techniques. This fault model represents the basis for the discussion in our paper.

In [**?**], the state model of each class in a system is augmented by taking into account the aspects, adding new states and transitions where necessary. A criterion is defined to evaluate test coverage on the augmented state model and to guide the developers in producing the test cases for the new states and transitions. The idea of incremental testing is mentioned as a way to limit the number of test cases to re-run on the weaved code, assuming the base code has passed all the tests.

In [**?**], AOP is used as a support for testing traditional code by expressing test adequacy criteria relative to crosscutting concerns. Aspects are used to obtain a fine grained instrumentation of the code and to enable dynamic monitoring of the testing coverage. Since existing pointcut models do not provide a granularity fine enough to support whitebox testing, authors defined their own pointcut model and language. A testing tool is provided and explained based on the proposed pointcut model.

In [**?**], data flow based unit testing for OOP code is extended to cope with AOP specific constructs. Control flow and data flow constraints are derived for the weaved code and a three level unit testing is presented to guide the developer while writing the test cases.

The authors of [**?**] analyze the impact of static crosscutting on the method binding. An algorithm for the calculation of the changed lookups is presented.

6 Future Works

In our future work, we will implement the AOP specific testing techniques described in this paper. Designator and composition coverage criteria will be evaluated on the field, once such an implementation is available. Moreover, we will consider also AOP specific testing techniques aimed at making existing and applicable approaches more effective for specific fault types.

Another direction of our future work will be the detailed definition and implementation of the AOP incremental testing method.

An important role in our future research will be given to the empirical assessment of the proposed approach to AOP testing, in comparison with the more traditional OOP testing methods applied to AOP code. Experiments will be performed trying to evaluate differences between them in terms of number and types of detected defects, test case complexity and test case definition effort.

References

[1] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, April 1997.