Aspects for Testing Aspects?

Dehla Sokenou, Stephan Herrmann
Technische Universität Berlin
Software Engineering Group
Sekr. FR 5-6, Franklinstr. 28/29, D-10587 Berlin
[dsokenou|stephan]@cs.tu-berlin.de

ABSTRACT

Within the field of aspect-oriented software development, testing can be discussed in two directions: (1) How can aspect technology help to build testing tools? (2) Which new requirements does aspect technology used in the system under test impose on the test strategy? In this paper we suggest to discuss both questions in combination. We discuss some prior work regarding each question and outline a perspective in which the application and also testing tools are smoothly developed using the same programming language and without the need for specialized instrumentation tools. From this perspective we derive some requirements concerning the aspect language used and also concerning the test method.

1. INTRODUCTION

In the past, aspect-oriented techniques have focused on aspect-oriented programming. At present, there are new research areas adressing aspect-oriented software engineering (AOSE). This includes aspect-oriented modeling and testing. Full synergy can be achieved by aspect-oriented modeling if models are also suitable for model-based testing of aspect-oriented.

Testing and aspects can be considered from two perspectives. First, aspects can be used for testing systems, generally object-oriented systems. Second, aspect-oriented systems must be tested.

So far, both perspectives have been investigated separately. In this paper, we look at some of the benefits of discussing both perspectives together. We present an approach for testing aspects using aspect technology, which raises some questions regarding not only the testing techniques and method but also the aspect language used.

In Section 2, we present testing problems, their aspect-oriented solution and the requirements for transferring the solution to aspect-oriented programs. We consider two aspect-oriented languages, AspectJ [2, 10] and ObjectTeams/Java [14, 8], and examine them with respect to the given requirements. In Section 3, we look at the aspect-oriented fault model in [1] to define requirements on the testing techniques used. Finally, in Section 4 we present our conclusions and give an outlook on future work.

2. TESTING AS A CROSSCUTTING CONCERN

Using aspect-oriented techniques for testing systems assumes that testing is a crosscutting concern with respect to the system under test (SUT). If testing is really a crosscutting concern, we should encapsulate all testing functionality in an aspect. This holds both for testing object-oriented systems and aspect-oriented systems.

In most cases, testing involves inserting additional code into the SUT. The new code is tightly coupled with the system under test and spreads across the whole system. This is typically known as scattering and tangling. Examples of crosscutting test code are:

- The SUT must be in a specific state before we can execute test cases. Normally, initialization and test-case execution are done by test drivers, which need privileged access to the SUT.
- The answer and state of the SUT must be observed and a verdict must be derived by a test oracle.

Both test drivers and test oracles must normally be removed after testing.

Adaptation of the SUT is classically done by automatically instrumenting the SUT using a specialized tool. Most of the tools focus on only one instrumentation problem, e.g. control flow coverage. Aspect-oriented programming is more flexible than classical instrumentation. Aspect languages offer the programmer many options that are otherwise available only when using sophisticated instrumentation tools. All instrumentation code can be collected in a small number of modules. The instrumentation may cover more than one problem. Aspects have privileged access to the adapted implementation, here the SUT. The original source code is not modified and aspects can easily be removed. Here, we see the advantages of using a load-time or runtime weaver instead of a compile-time weaver because deployment is facilitated if compiled code is not modified.

Given these capabilities, an aspect language may replace much of the technology used in classical instrumentation tools. Thus, tools are more light-weight and easier to develop. As a result, we envision libraries of testing aspects, which can easily be adapted to the needs of each specific project, i.e. a project can define its tailored testing method and compose the supporting tool suite from such reusable aspects. For these reasons, there is a lot of research on aspect-oriented programming for testing (see, for example, [4, 12, 15, 3, 16]).

On the other hand, using aspects for testing is not yet a complete solution. Today, aspect-oriented languages do not allow us to insert code per line or statement which is needed for, say, source code coverage criteria. A flexible instrumentation technique requires testing tools to support the tester in all areas covered by traditional testing tools.

If testing is a crosscutting concern, why don't we try to transfer successful approaches from the field of object-oriented testing with aspects to aspect-oriented testing? This means testing aspects with aspects.

Although it may seem easy to transfer object-oriented testing techniques to aspect-oriented systems, we are faced with several problems. How can we adapt aspect-oriented programs? Will application aspects and testing aspects interfere? Can application aspects be tested by testing aspects?

In the following sections, we outline solutions for testing object-oriented systems using aspect-oriented techniques. We then define requirements for aspect-oriented languages which allow the testing of aspects with aspects.

2.1 Test Oracles

Most of the related work focuses on realizing test oracles with aspects. Popular approaches are checking pre- and post-conditions and class invariants [15, 3] and implementing state-based test oracles with aspects [4, 16]. Another approach focuses on integrating mock objects into the SUT [12].

Here, we concentrate on checking contracts and states with aspects, two of our own research topics.

2.1.1 Example: Design by Contract

To support testing, a program can be extended by pre- and post-conditions and class invariants. These parts of a contract between client and server are typically part of the specification and can be checked at runtime. The object-oriented language Eiffel has built-in support for contracts which can be checked at runtime. However, such a capability is lacking in main-stream programming languages. Here, an aspect-oriented language can be used to check the pre-condition before a method is called, and check the post-condition after a method has been executed. Class invariants must be checked before and after method execution.

When subtype polymorphism comes into play, the contract of any subclass must ensure substitutability with respect to its superclass. A client making any assumptions based on the static type of a supplier must remain valid if a method call is actually answered by an object whose dynamic type is a subtype of the expected type. The rules of subcontracting ensure exactly this required conformance.

Figure 1 shows this situation. The client only has to ensure the pre-condition of the supertype method, but the pre-condition of the subtype method must also hold. The subclass server must ensure its own post-condition but the client expects the post-condition of the superclass server.

Regarding aspects, the contracts must be extended (see Figure 2). Aspect code should have its own pre- and post-conditions. Conceptually, this calls for a chain of checks, as pointed out in [13]: the checking of a method's pre-condition, e.g., has to be split into the client side, where the pre-condition of the static receiver type is checked and, just before entering the actual method code, the pre-condition associated to the dynamic receiver type must be checked. Between these two checks, an arbitrary number of before advices may intercept the control flow, each having its own pre- and post-conditions. These steps are important in order to assign the blame to the right party whenever any assertion is violated.

The distinction between call and execution join points is an important step toward weaving assertion checks into the correct points within the control flow. However, adding assertions to an existing aspect by means of an additional testing aspect reveals a weakness in current join point languages: these languages (as part of an aspect language) focus on identifying points in the control flow of ordinary classes. Support for aspects of aspects is mostly insufficient. Even the newly introduced keyword `adviceexecution` of AspectJ does not provide a complete solution because it does not allow us to attach an aspect to a specific advice. This closely relates to orthogonality issues as discussed in [7]. The authors critize that new language features of aspect languages cannot be applied freely to each other. The lack of pointcuts for a specific advice is an example of this lack of orthogonality.

In ObjectTeams/Java, the situation is simpler because no distinction is made between method and advice. All aspect code is written in named methods, too. Thus, there is no problem with attaching an aspect to a given piece of aspect code.

2.1.2 Example: State-Based Testing

Aspects can be used to implement statecharts as test oracles and to integrate them into the SUT. In [4], a statechart hierarchy is flattened before statecharts are integrated into the system under test. Our approach (see [16]) supports statechart hierarchies without flattening, the history connector, and parallelism. It does so by using the programming language ObjectTeams/Java, which allows aspects to be activated and deactivated at runtime. We consider dynamic aspect activation and deactivation as essential for efficient state-based testing. The capabilities of ObjectTeams/Java for controlling aspect activation have proved highly suitable for this task.

If we wish to transfer state-based testing techniques to aspect-oriented systems, we first have to specify aspect-oriented systems with statecharts or other state machines.

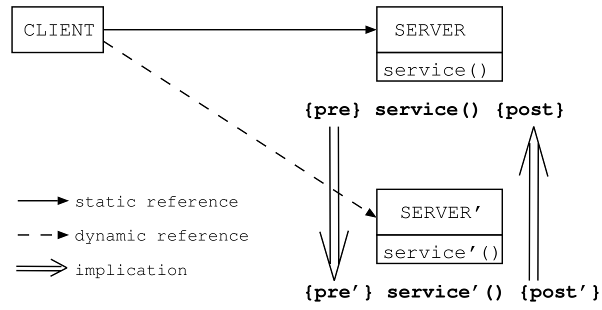

Figure 1: Design by contract in object-oriented systems.

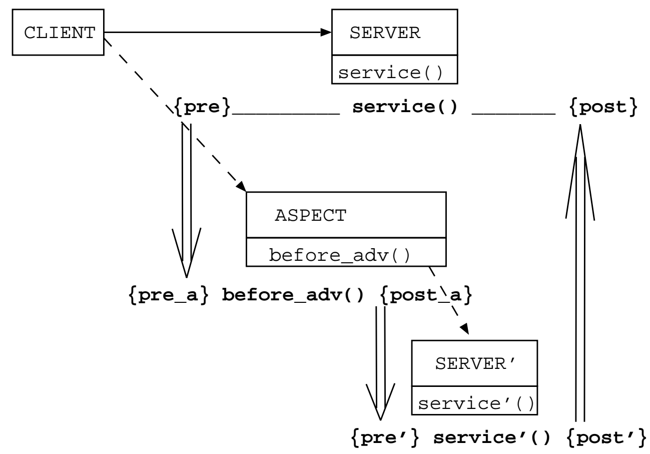

Figure 2: Design by contract in aspect-oriented systems.

An extension of statecharts by aspects is not needed if we consider aspects only as a technical realization of a problem and statecharts as a specification of the input/output behavior with hypothetical states (black-box view). In this case, we consider statechart models independently of implementations. The work on event-based aspect-oriented programming [6] hints at similarities between aspect-oriented programming and trigger-action systems[1]. Thus, the gap between statecharts and an implementation may actually decrease when using aspect-oriented programming. On the other hand, state machines can be extended by aspect-oriented features, as in the state-based approach to testing aspect-oriented systems presented in [17]. Since object-oriented statecharts are closely related to object-oriented systems (transitions are triggered by method calls), aspect-oriented statecharts can be closely related to aspect-oriented systems triggering transitions by pointcuts.

The requirements for the aspect-oriented language are similar to the ones defined in the design by contract section because the state of the SUT must be checked before and after a state-changing event, generally a method call. So, again, we need pointcuts of pointcuts and/or advices.

2.2 Test Driver

Besides implementing test oracles, we have experimented with aspect-oriented test drivers. Our test drivers consist of two modules: a regular OO program, which establishes initial states for tests and aspects which perform suitable tests on an initial state. In our example, the pointcut is every creation of an object in a class hierarchy. If an object of the superclass or one of its subclasses is created, it is tested with the superclass test suite. The test-driver aspects ensure that only one object of each type is tested.

Similar approaches are possible for testing aspects. Class hierarchies are similar to aspect cascades. In object-oriented systems, a server can be accessed by a client of the declared type or some of its subtypes. A client can access a server of the declared type or some of its subtypes. In aspect-oriented systems, a base object must behave correctly with or without aspects and an aspect must behave correctly in all contexts defined by the pointcuts.

One question is: Can we develop test drivers that can test aspect cascades? A test driver must know which aspects have already been tested with which base objects. This is not a trivial problem. We must know which join points are collected by a pointcut and we must calculate the difference between all join points and those already executed. And how can we force the application to execute specific advices, in particular regarding dynamically activated aspects or aspects depending, e.g., on control flow? A complete aspect-oriented control-flow analysis is needed.

[1] In [9] we show, how the language ObjectTeams/Java is extended with guard predicates in order to realize the full separation into events, *conditions* and actions (ECA). The scheme of ECA-triplets has proven useful as part of the statecharts formalism. The same concept can now be applied even at the programming language level, which we are currently exploring for implementing test-oracles as presented in [16].

A second question is: What is a minimal criterion for covering aspect-oriented SUT? Is covering all aspects enough, or must every aspect be tested with all base applications and the base application with all aspects? In object-oriented languages, not all possible control flows are tested. In most cases, we do not test all subclasses on the client side with all subclasses on the server side. The combinatorial problem of testing is reduced using a fault model or heuristics. An aspect-oriented fault model is presented in [1] (see Section 3). Heuristics on typical and common faults in aspect-oriented systems are not yet available.

A third question is: How can we switch between test initialization and test execution? For this task the capability to explicitly activate an aspect during program execution is very handy. In AspectJ, a test driver could switch to test modus with a cflowbelow pointcut like `!cflowbelow( call(*.testInit*(..))`. However, this pointcut would have to be part of *every* pointcut used for integrating the testing aspect. For this kind of switching explicit activation is more modular.

3. ASPECT-ORIENTED FAULT MODEL

Requirements for aspect-oriented languages can also be extracted from the aspect-oriented fault model presented in [1]. There, we find six classes of aspect-oriented faults. Our question is whether an aspect-oriented language can avoid the cause of these faults. Increasing the language strength can reduce failures (much as a typed language reduces type failures at runtime). We believe that increasing language strength can help avoid some of the faults resulting from incorrect aspect precedence or changes in control dependencies. In some cases, the SUT can only be tested if the aspect-oriented language supports the testing strategy, e.g faults resulting from incorrect strength of join points.

Incorrect strength of join points. A pointcut can be specified too strong or too weak. In the first case, not all necessary join points will be collected. In the second case, join points are collected that should be ignored. Faults resulting from the incorrect strength of pointcuts can be found if we can decide which join points are already covered by the test (see also the test driver section in Section 2).

Incorrect aspect precedence. The order of woven aspects can affect system behavior. If aspects are not independent, an aspect precedence must be defined. Independent aspects can be executed in any order. In a program with an undefined aspect order, all aspect orders may have to be tested. An undefined order might be acceptable to the programmer, but for testing this opens up new combinations that would have to be tested. Thus, from the testing perspective, each program should have an aspect order that can be determined from the program, in order to avoid unnecessary tests.

Failure to establish expected post condition. The client of an object does not know that the server is adapted by one or more aspects. The post-condition it expects is the post-condition of the server objects

method (see also the design by contract section in Section 2). The problem is similar to subcontracting. The pre-condition of a subclass's method must be weaker than or equal to that of the superclass's method, and the post-condition stronger than or equal to that of the superclass's method. Thus, in languages like Eiffel, both pre-conditions are combined by a logical OR, and both post-conditions by a logical AND. [13] conciders precisely what implications must hold when combining contracts of aspects with the contract of the original method. Thus, the design by contract testing approach can indeed be extended for aspect-oriented programs.

Failure to preserve state invariants. An aspect has access to attributes of its adapted objects. Access to attributes can violate state invariants. Here, we recommend a state-based testing approach like the one presented in [16]. States of the adapted objects have to be tested without knowledge of aspects.
Testing can, of course, be reduced if static analysis yields that an aspect only reads but never changes the state of its base object(s). Thus, the distinction between observers and assistants ([5]) can be of great help for testing, too.

Incorrect focus on control flow. Often, a decision about pointcut selection or advice activation is needed at runtime. Dynamic aspect activation and join point selection can cause faults that are difficult to find. Although languages with static aspects and static join point selection have advantages here, we do not want to restrict our approach to static aspect-oriented languages. Instead, better tools are needed. It already helps the testing tool if reflection mechanisms exist that enable a list of active aspects to be obtained for analysis. In ObjectTeams/Java, for example, such a capability can easily be constructed by intercepting all calls to `activate` and `deactivate`.

Incorrect changes in control dependencies. When using an around advice, changes in control-flow dependencies can be incorrect, e.g., if a proceed call is lacking. Control-flow analysis can help to avoid or detect these faults. If proceed is not called, the developer should be notified in form of a warning. ObjectTeams/Java generates such a warning if a call of the base method is missing in any branch of the control flow within a `callin replace` method[2]. The same holds for (potentially) duplicate base calls.

The next section summarizes our thoughts on testing aspects using aspect-oriented techniques and our requirements for aspect-oriented languages to support testing.

4. CONCLUSION

We see considerable advantages in using aspects for testing aspects. Aspect-oriented techniques are well suited when testing object-oriented systems, so why step back when testing aspect-oriented systems? But to transfer aspect-oriented testing techniques to aspect-oriented SUTs, we need languages that smoothly support aspects of aspects.

[2]A `replace` method is similar to an `around` advice in AspectJ.

Requirements for aspect-oriented languages can be summarized as follows:

- A flexible join point language is needed to encapsulate general testing purposes in one aspect.
- We need join points for advices to instrument aspects as well. Generally, testing aspects using aspects emphasizes the importance of language orthogonality, as is discussed in [7].
- Aspect activation at runtime can help to implement dynamic test support, like state-based test oracles. It can also help to separate the initialization phase of testing from actual test execution.
- Reflection mechanisms have to be extended so that information about actual join points, pointcuts and advices can be obtained.
- A good control-flow analysis by the compiler can reduce faults resulting from incorrect changes in control dependencies, e.g. from forgotten proceed statements.
- Code weaving at load- or runtime facilitates deployment of the SUT because it avoids maintaining two versions of the same software, one with testing aspects and another without.

ObjectTeams/Java seems to have advantages for testing aspects with aspects because aspects can have other aspects as their base and explicit aspect activation can be controlled from within the program. Deployment is facilitated by using a load-time weaver. See [11] for a successful example of the use of load-time byte code adaptation for testing. The work presented uses the same framework —JMangler— as is used by the runtime enviroment of ObjectTeams/Java.

However, the current version of ObjectTeams/Java does not yet have a sophisticated join point language like AspectJ. Thus, AspectJ has advantages for building general test aspects. Also, AspectJ has changed the weaving strategy from source-code to byte-code weaving. Support for load-time weaving is emerging, but it is not yet clear how much of the deployment issue is solved by this technology.

However, neither language supports code instrumentation per line or statement.

Our future research will focus on investigating better testing techniques based on aspect-oriented programming techniques. We wish to transfer these techniques to both object-oriented and aspect-oriented languages. Our goal is to develop testing tools that support object-oriented as well as aspect-oriented parts of a system under test.

When analyzing the different goals and requirements of tools for test instrumentation and current aspect languages, there is still a chance that full integration of these two techniques might prove unfeasible. For example, per-sourceline aspects might not be desirable for general-purpose aspect languages, but they would be needed for instrumentation using aspects. In that case, we suggest developing a shared tool platform

that provides the needed low-level facilities. Test instrumentation and aspect languages could then simply be seen as different front-ends to the same machine. So far, it is not yet clear whether this common platform would just be a regular meta object protocol or whether more specific commonalities will prevail.

Despite this question, we believe that aspects can be used for testing aspects. Applying techniques designed for testing object-oriented systems to aspect-oriented systems is a step toward developing testing techniques for aspect-oriented systems. However, some of these techniques must be adapted to the special features of aspect-oriented programs.

5. REFERENCES

[1] R. T. Alexander, J. M. Biemann, and A. A. Andrews. Towards the Systematic Testing of Aspect-Oriented Programs. Technical report, Colorado State University, Department of Computer Science, USA, 2004.

[2] AspectJ-Homepage. htpp://www.aspectj.org.

[3] L. C. Briand, W. Dzidek, and Y. Labiche. Using Aspect-Oriented Programming to Instrument OCL Contracts in Java. Technical report, Carlton University, Canada, 2004.

[4] J.-M. Bruel, J. Araújo, A. Moreira, and A. Royer. Using Aspects to Develop Built-In Tests for Components. In *AOSD Modeling with UML Workshop, 6th International Conference on the Unified Modeling Language (UML)*, San Francisco, USA, 2003.

[5] C. Clifton and G. T. Leavens. Observers and Assistants: A Proposal for Modular Aspect-Oriented Reasoning. In *FOAL 2002: Foundations of Aspect-Oriented Langauges (AOSD-2002)*, pages 33–44, Mar. 2002.

[6] R. Douence, O. Motelet, and M. Südholt. A Formal Definition of Crosscuts. *Lecture Notes in Computer Science*, 2192:170–184, 2001.

[7] K. Graversen and K. Østerbye. Aspects of Aspects — a Framework for Discussion. In *European Interactive Workshop on Aspects in Software*, 2004.

[8] S. Herrmann. Object Teams: Improving Modularity for Crosscutting Collaborations. In *Objects, Components, Architectures, Services, and Applications for a Networked World (Net.ObjectDays Conference)*, volume 2591 of *Lecture Notes In Computer Science*, Erfurt, Germany, 2002. Springer.

[9] S. Herrmann, C. Hundt, K. Mehner, and J. Wloka. Using Guard Predicates for Generalized Control of Aspect Instantiation and Activation. In *Dynamic Aspects Workshop (DAW'05), at AOSD 2005*, Chicago, 2005.

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)*, volume 2072 of *Lecture Notes in Computer Science*, Budapest, Hungary, 2001. Springer.

[11] G. Kniesel and M. Austermann. CC4J - Code Coverage for Java - A Load-Time Adaptation Success Story. In *Proceedings of the* 1st *IFIP ACM Working Conference on Component Deployment*, number 2370 in LNCS. Springer, 2002.

[12] N. Lesiecki. Test Flexibility with AspectJ and Mock Objects. http://www-106.ibm.com/developerworks/java/library/j-aspectj2/?loc=j.

[13] D. Lorenz and T. Skotiniotis. Contracts and Aspects. Technical Report CCIS-03-13, Northeastern University, Boston, 2003.

[14] Object Teams-Homepage. http://www.objectteams.org.

[15] M. Richters and M. Gogolla. Aspect-Oriented Monitoring of UML and OCL Constraints. In *AOSD Modeling With UML Workshop, 6th International Conference on the Unified Modeling Language (UML)*, San Francisco, USA, 2003.

[16] D. Sokenou and S. Herrmann. Using Object Teams for State-Based Class Testing. Technical report, Technische Universität Berlin, Fakultät IV - Elektrotechnik und Informatik, Germany, 2004.

[17] 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.