On Testing Crosscutting Features using Extension Join Points

Roberta Coelho[1], Vander Alves[2], Uirá Kulesza[1] , Alberto Costa Neto[2], Alessandro Garcia[3], Arndt von Staa[1], Carlos Lucena[1], Paulo Borba[2]

[1]*Departamento de Informática – Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio)*
{roberta, uira,lucena,arndt}@inf.puc-rio.br
[2]*Centro de Informática – Universidade Federal de Pernambuco*
{vra, acn, phmb}@cin.ufpe.br
[3]*Lancaster University, Computing Department, Lancaster - United Kingdom*
garciaa@comp.lancs.ac.uk

Abstract

Recently, one arguing question in the context of product line development is how to improve the modularization and composition of crosscutting features. However, little attention has been paid to the closely related issue of testing the crosscutting features. This paper proposes an approach for testing the crosscutting features of a product line based on the use of a previously proposed concept called Extension Join Points (EJPs).

1. Introduction

Framework technology has been widely used in the development of software product lines (PL) as a way of enabling systematic reuse-in-the-large. OO frameworks allow feature[1] modularization and composition, and offer extension options to target applications. Besides their advantages, some researchers [5, 19, 24] have recently described the inadequacy of OO mechanisms to address the modularization and composition of many framework features, such as, optional, alternative and crosscutting composition features.

Crosscutting features represent concerns that are not well modularized in OO implementation. They are often spread over several modules of a software system and tangled with other features' implementation. Examples of such features are: security and transaction management. Hence, it is difficult to write a feature unit test for such features since there is no specific unit to be tested [7,16].

Aspect-oriented software development (AOSD) [10, 11] has emerged as a technology which aims at improving the modularization of crosscutting concerns. Recent work [2, 13, 14, 21] have been exploring the use of aspects to improve the modularization of crosscutting features in product lines.

While AOSD provides an effective way for modularizing crosscutting concerns and consequently providing a "unit" upon which a unit test can be defined, it brings new challenges to software testing. The new programming constructs provided by aspect-oriented languages are sources for new types of programming faults. Alexander et al [1] defined an initial candidate fault model for AOPs with new classes of AOP-specific faults, in addition to faults that can exist in object-oriented systems such as Java.

In a previous work [14], we have presented an approach to systematize the extension of OO frameworks by means of aspects. Aspects are used to modularize optional, alternative and integration crosscutting features encountered in the implementation of OO frameworks. The aspects introduce crosscutting features in the framework core by means of extension join points. In this paper, we propose a complementary verification strategy used to test the crosscutting features implemented by variability and integration aspects.

The remainder of this paper is organized as follows. Section 2 presents background by showing the basic concepts of AOSD and revisiting some research work on product line testing. Section 3 discusses briefly our framework development approach based on AOP and extension join points (EJPs). Subsequently, Section 4 illustrates a case study in which EJPs were implemented using AspectJ. Section 5 presents our test strategy for crosscutting features that relies on the use

[1] A feature is a system property that is relevant to some stakeholder and is used to capture commonalities and discriminate among systems in a family [9].

of EJPs. Some discussion and related works are presented in Section 7. Finally, Section 8 summarizes our contributions and provides directions for future work.

2. Background

This section briefly revisits the basic concepts of AOSD and research work on product line testing methodologies.

2.1 Aspect Oriented Software Development

Aspect-oriented software development (AOSD) [10,11] supports the modularization of crosscutting concerns by providing abstractions to extract these concerns and later compose them back when producing the overall system. Such abstraction is called aspect.

AspectJ [4] is an implementation of AOP for the Java programming language. The aspect abstraction in AspectJ is composed of: inter-type declarations, pointcuts and advices. Inter-type declarations specify new attributes or methods to be introduced in specific classes. Joinpoints are well-defined locations within the base code where a concern can crosscut the application. Examples of join points are method calls and method executions. Pointcuts have a name and are collections of join points. Advices are a special method-like construction of aspects which are used to attach new crosscutting behaviors along the aspect pointcuts.

2.2. Software Product Line Testing

According to product line testing methodologies proposed so far [20, 22, 12], product line testing should be done at three main levels: at unit (or component) level, at integration (or feature) level, and at system level. Features are a suitable integration criteria since the instances of a product line often differ basically in the availability of product line features.

Current approaches propose general guidelines to structure the whole process of product line testing. For instance, they state that the tests defined for the core assets, at any one of these levels, should be treated as core product line assets and managed consistently; and that such tests may be fully or partially reused across versions of the product line and at specific products. However, there is still a lack of techniques to help developers in the low level design of tests at each level.

This work proposes a feature-level testing approach for features that spread over several modules of software system. It is based on our approach to framework extension based on aspect-oriented techniques. Next section describes our approach.

3. A Framework Extension Approach

In a previous work [14], we have proposed a systematic approach for framework extension by means of aspects. In our approach, we defined the concept of Extension Join Point (EJP). The EJP consists on a unified way of designing and documenting existing crosscutting extension points. It provides new means for extending framework core functionality, introducing optional and alternative crosscutting features, and integrating the framework elements with other components and frameworks. EJP represents a new kind of framework hotspot, different from the well-known object-oriented extension points. Figure 1 illustrates these two kinds of framework hotspots.

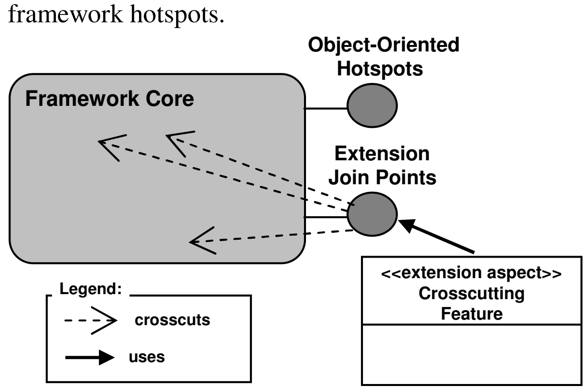

Figure 1: OO hotspots x Extension Join Points.

The object-oriented hotspots are usually represented as abstract classes that should be extended, or interfaces that should be implemented during framework instantiation. On the other hand, the EJPs represent framework hotspots that will be used by aspects that will implement a crosscutting feature in the framework. We call extension aspects, the aspects that address the implementation of a crosscutting feature, as shown in Figure 1.

The EJPs are inspired by a recent study performed by Sullivan et al [25] proposing the use of an interface between the base code and the aspects, called crosscutting interfaces (XPIs). EJP extends the concept of XPI to the context of framework development. The EJP comprises different attributes from the ones proposed by Sullivan in the XPI specification. It also defines a set of internal and extension contracts which regulates the relationships between the framework and the extension aspects (Figure 1).

Each EJP is composed of the following elements: (i) a name that is represented by the aspect's name; (ii) a scope which defines all the framework elements that are "encapsulated" by the EJP; (iii) a set of crosscutting extension points, which specifies the framework join points that represent relevant events or transition states occurring during the execution of the framework functionalities; and (iv) a set of internal and extension contracts.

The framework internal contracts define constraints whose purpose is to assure that framework refactorings and evolution do not affect the functionality of its extension aspects. They are classified in the following categories:

- *Structural*: which aims to guarantee the framework implements specific interfaces defined by the EJPs; and
- *Behavioral*: which assures the framework EJPs comprises all and only the framework events (or states) that the EJP is intended to expose.

The framework extension contracts are used to assure that each extension aspect respects constraints and invariants of the framework. The following categories were defined:

- *Structural:* these contracts assure that aspects only extend the framework join points exposed by the EJPs, and specify the framework classes methods that can be invoked by the extension aspects; and
- *Behavioral:* define specific pre- and pos-conditions that must be preserved before and after the execution of extension aspect advices.

Tables 1 and 2 show different AspectJ mechanisms that we have used to implement these contracts.

Contract Type	AspectJ Implementation
Structural	Specification of interfaces that must be implemented by framework classes. The obligation to implement these interfaces is assigned by the EJPs using the `declare parents` inter-type construction of AspectJ. The interfaces are also declared inside the aspects that represent the EJPs.
Behavioral	Implementation of enforcement policies guaranteeing that the extension join points are called only and in all appropriate places inside the framework. This contract can be specified using `declare warning` and `declare error` AspectJ statements.

Table 1. Framework Internal Contracts.

Contract Type	AspectJ Implementation
Structural	It is possible to define AspectJ contract to restrict the framework classes' methods that can be accessed inside the extension aspects. There are two different ways to specify it: (i) using `declare warning` and `declare error` AspectJ statements, which allow the static verification of policies; and (ii) by defining advices which intercept every advice execution that realizes calls to the framework classes' methods. The `adviceexecution()` pointcut designator is used to intercept the advices execution.
Behavioral	This contract defines pre- and pos-conditions that must be assured before and after the advice execution. These contracts are also defined using `adviceexecution()` pointcut designator to intercept the advices execution.

Table 2. Framework Extension Contracts

Due to a current limitation of the AspectJ it is not possible to automatically assure that aspects only extend the framework join points exposed by the EJPs. Hence, to assure it, the developers must follow the programming practice of using only pointcuts specified in the EJP[2], which will be checked during manual inspections.

4. Case Study: J2ME Game SPL

In this case study, we implemented variant features of an industrial J2ME game software product line (SPL) based on EJPs. J2ME games are mainstream mobile applications of considerable complexity [2]. Their overall structure and behavior are defined by a framework known in this domain as the *game engine*.

Essentially, this is a state machine whose state change is driven by elapsed time and user input through the device keypad. State changes affect the state of various drawing objects (*game actors*) and how they interact. Then, these objects are drawn again after such state changes. Typical hot-spots of this framework include some abstract classes defining basic drawing capability for game actors.

We have defined EJPs in order to allow the composition of crosscutting extensions in its basic functionality. Some interesting EJPs are the following: (i) images initialization usage events; (ii) the drawing of specific images; and (iii) game startup and changing screens events. We have chosen these EJPs because they represent relevant events that can be of interest when extending the game engine core workflow. The resulting SPL architecture is shown in Figure 2.

In Figure 2, package `rain.core` denotes the SPL core, i.e. the game engine. Package extension join points encapsulate all the EJPs, which are used by variability aspects in corresponding packages to implement crosscutting extensions. For example, the `Clouds` variability aspect implements the feature of clouds rolling in the game screen background. This

[2] Larochelle et al [15] have proposed a mechanism, called join point encapsulation, which aims to prevent selected join points from being modified by aspects. Since this mechanism was implemented only to previous versions of AspectJ, we did not have the chance to experiment it in our case studies.

aspect relies on the `CloudEvents` and `ResourceEvents` EJPs, which expose the drawing of scenery objects and image loading. The `FlipBase` variability aspect and their extending aspects provide the features of manually and automatically flipping an image, for devices that do not have and for those that have built-in flip API, respectively. Those aspects depend on the `DrawingEvents` and `ResourceEvents` EJPs, which specify the drawing of images of game actors.

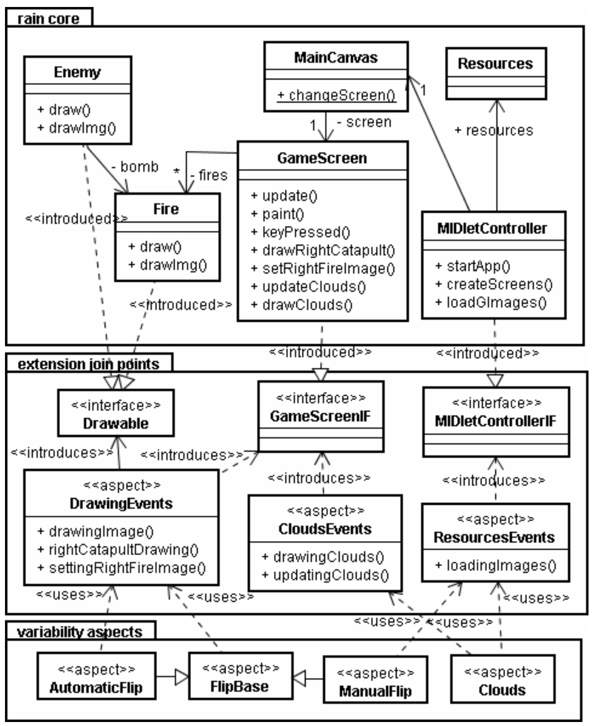

Figure 2: Architecture of J2ME Game Product Line.

This case study is used in the next section in order to exemplify some test steps of our crosscutting feature testing strategy.

5. Crosscutting Feature Testing Strategy

This section presents a unit test approach for the crosscutting features of a product line. A crosscutting feature can be broken down in two major parts: what the feature does and where it is applied. The faults discovered during a crosscutting feature test can arise from one of the following sources[3]: (i) bugs in the crosscutting feature logic; (ii) inaccurate pointcut designator which intercepts a wrong set of join points; (iii) an emergent property created by interactions between the crosscutting features and the base code; and (iv) faults in the core components themselves. We propose a strategy for testing crosscutting features based on the use of EJPs, composed by five steps, as illustrated in Figure 3.

One of the hardest issues about testing a crosscutting feature is to check whether the places where the crosscutting features apply are correct. Checking all the places affected by a crosscutting feature can be a daunting task, specially if we need to check its negative scope (i.e check whether there is any accidental inclusion of a point to be intercepted). To help developers in finding faults on pointcuts, we supplemented our test strategy with steps of manual and automatic inspections based on the use of EJPs (Steps 1 and 2).

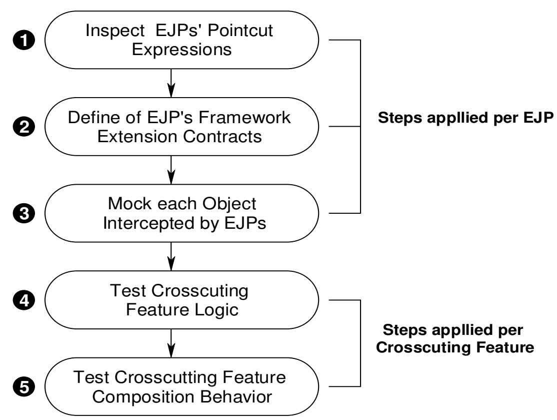

Figure 3: The steps from the crosscutting feature testing approach.

This feature-level testing approach aims at finding faults of types (i) (ii) and (iii) defined previously. Steps 1 and 2 aim at finding faults of type (ii). Step 3 implements a test infrastructure to be used in Step 4 which detects faults of type (i). Finally, Step 5 looks for bugs of kind (iii). In order to detect faults of type (iv) traditional OO testing techniques can be applied to the PL context.

Step 1: Inspect EJPs' Pointcut Expressions

For each EJP defined for the system, the developer should check whether the EJP pointcuts intercept the correct places and only them. In order to accomplish this, the developer can use the crosscutting visualization tools available at AJDT (the most used AOP IDE). It can also use string matching algorithms to calculate the distance between the pointcuts and the join points in the system and verify whether there is

[3] There are faults that emerge from a property created when more than one aspect affects the same element in the base code. However, this kind of fault can not happen in a feature unit test scenario, in which one feature is tested at a time.

any accidental inclusion of a point to be intercepted [16, 3].

This step is very time consuming since there is not a fully automatic way to detect this kind of faults [16]. According to our strategy, however, only the EJPs have to be inspected and the crosscutting features only need to reuse them. If the EJPs were not used to mediate the relationship between the core and the extension aspects, every extension aspect would have to be inspected in previous step, thus compromising its scalability.

The techniques presented at this step are not sufficient to verify pointcut expressions that involve complex dynamic conditions, which depend on the execution stack. Such expressions can only be verified after weaving. Steps 4 and 5, detailed next, can help developers in detecting such kind of faults.

Step 2: Define EJPs' Framework Extension Contracts

Since we have inspected the set of pointcut expressions defined by the EJPs, now we need to assure that each extension aspect will respect the constraints and invariants of the framework. In order to do it the developer should define a set of EJP's framework extension contracts.

As detailed in Section 3, these contracts can be checked during manual inspections, verified at compilation time or runtime. The EJP's contracts evaluated in runtime act as test oracles, since they will alert the developer when a contract is violated during feature-test executions. We can exemplify one of such runtime Extension Contracts in our J2ME Game PL case study. Figure 4 illustrates a contract associated to the `DrawingEvents` EJP shown in Figure 3.

This contract states that the crosscutting extension features - represented in this implementation by aspects in the `variabilityaspects` package - are not allowed to access framework elements besides the `Graphics` and `Drawable` types. The `FWScopeNotAllowed()` pointcut denotes calls to framework internal types, where we assume that the `Drawable` interface and the `Graphics` class should be visible to the extension aspects. The `aspectsPackages()` denotes calls within such aspects.

The PL developers can define extension contracts as complex as needed. We are currently investigating simpler ways of defining interaction rules, thereby not requiring the developer to learn too complex AspectJ constructs.

```
public aspect DrawingExternalContractChecker {
   // Framework Scope - Calls Not Allowed
   public pointcut FWScopeNotAllowed():
         call ( * !(Drawable+||Graphics).*(..) )
         && call (* raincore.*.*(..));

   public pointcut aspectsPackages():
                  within(variabilityaspects..*);

   declare warning:
       FWScopeNotAllowed() && aspectsPackages():
       "Extension aspects are accessing \
        internal framework details";
}
```

Figure 4: Interaction Contract checking.

Step 3: Mock each object intercepted by the EJPs

Mackinnon et al [18] proposed the Mock Object test design pattern[4] [6], and since then, Mock Objects have been recognized as a useful approach to the unit test and design of object-oriented software. A Mock Object is a regular object that acts as a stub, but also includes assertions to instrument the interactions of the fake object with its neighbors. The Mock Object allows the unit test of a component that depends on others which may not be implemented yet. This is exactly the scenario that we find when testing a crosscutting feature that affects hotspots that will only be implemented by PL products.

This third step states that the PL developer should define mock objects of the code intercepted by the EJPs. Both the real object and its mock version should implement the same interface. Since the EJP only refers an object by its interface, it can remain ignorant to whether it is intercepting the real object or the mock object. The affected code can be a core asset or a framework object-oriented hotspot.

One mock object should be created for each interface intercepted by the EJP. In the J2ME case study we implemented mock objects for the following interfaces: `Drawable`, `GameScreenIF` and `MIDletControllerIF`.

Step 4: Test the Crosscutting Feature Logic

The set of Mock objects defined in the previous step is used at this one to enable the testing of crosscutting features logic. At this step the crosscutting feature is weaved with one or more mock objects and each method of the resultant component is unit tested.

The Mock Object can simulate some error conditions (i.e throw of an exception) on the base code [18]. As a consequence, it allows the developer to test the crosscutting features under abnormal conditions. Some crosscutting features, when exposed to abnormal

[4] A Test Design Pattern defines a good design solution for a system intended to test another system.

conditions, execute statements that throw exceptions, and as a consequence might cause undesired modifications in the system control flow.

To improve the testability of crosscutting features, the developer can also split each crosscutting feature in two different components [16]: (i) one that comprises the feature's logic; (ii) and other that specifies where these features should be applied - the pointcuts. Doing so, the developer can use well known OO testing criteria to test crosscutting features' logic: statement coverage, branch coverage, condition coverage, and dataflow coverage [23].

Sometimes, however, it is difficult to extract the logic from the crosscutting feature (i.e when crosscutting features include new methods and attributes in the affected object - intertype declarations), in such cases the crosscutting feature is only tested after the weaving process.

Step 5: Testing Crosscutting Feature Composition Behavior

The crosscutting feature composition behavior results from the interaction between the crosscutting feature and the base code (arises after weaving) [17]. This step aims at checking the behavior of functionalities affected by the crosscutting feature against their specification, as if the developer would do if the crosscutting feature code were scattered among affected features. The test should fail if the crosscutting feature misbehaves or does not apply at the specified points.

This kind of tests reveals faults that just occur when the features interact. However, it is difficult to diagnose a failure detected in such tests, because the cause can be in the crosscutting code, in the base code, or the crosscutting code not being applied in the appropriate place.

According to Colyer et al. [8] the crosscutting concerns should be classified as: orthogonal, altering, and stateful. Orthogonal aspects do not change control or data dependencies in the system (i.e logging). Altering aspects change control flow or data flow of a system. (i.e aspects using around advice). A stateful aspect has behavior that depends on an aspect attribute or introduced object attributes. Thus, at this step, the developer does not need to test crosscutting feature composition behavior for the orthogonal aspects. Because, it was already addressed by the step 4.

6. Discussions and Related Work

As was shown at Step 5, when a non-orthogonal crosscutting concern is weaved with the base code it modifies the base control structure and behavior. As a consequence, existing test suites may be missing in covering the crosscutting composition behavior. Thus, a new set of test cases needs to be defined to each feature, affected by a non-orthogonal aspect, in order to cover the resulting crosscutting behavior. This process is costly; however, the test suites will be used during the tests of each PL product.

In order to determine which test cases must be re-run when a crosscutting feature is added to the base code, the tests should be treated as core product line assets and managed consistently.

Research on testing aspect-oriented programs [26, 27, 3] has been focused on code-based unit and integration testing, automated test case generation, and fault model used for testing. Some of these works can be used in our testing approach. For instance, Xie et al have proposed a framework for generating test inputs for AspectJ programs [26], Step 4 could be extended in order to incorporate the test generation solution proposed by them. Zhou et al. [27] have proposed an algorithm based on control flow analysis for selecting relevant test cases, this technique could be applied on Step 4 and 5 in order to select the test cases to be executed.

8. Conclusions and Future Work

In this work, we have proposed a systematic approach for the testing of OO systems that have many of their crosscutting features implemented by means of aspects. In particular, our approach is complementary to a framework development approach proposed previously, which addresses the modularization of optional, alternative, and integration framework crosscutting features by using AO techniques. Our testing approach is composed of five complementary steps (Section 5) ranging from aspect pointcuts inspections to the unit test of every crosscutting feature using mock objects. Moreover, a set of contracts can be defined to guarantee an adequate interaction between the framework core, the EJPs and the extensions aspects. Thus, different ways of errors can be detected by using these different mechanisms of software testing.

As our approach is still under development, we intend to refine it by addressing the testing of different software product lines or software family architectures implemented using aspects. Also, the development of a testing tool supporting the generation of many elements (mocks, aspect unit testing) from the approach is under investigation.

9. References

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

[2] Alves, V., Matos, P., Cole, L., Borba, P., Ramalho, G., Extracting and Evolving Mobile Games Product Lines. Proceedings of SPLC'05, LNCS 3714, pp. 70-81, September 2005

[3] Anbalagan, P. and Xie. T., APTE: Automated Pointcut Testing for AspectJ Programs. Proc. of the 2nd Workshop on Testing Aspect-Oriented Programs (WTAOP 2006) (to appear).

[4] AspectJ Team. The AspectJ Programming Guide. http://eclipse.org/aspectj/.

[5] Batory, D., Cardone, R. and Smaragdakis , Y., Object-Oriented Frameworks and Product-Lines. 1st Software Product-Line Conference (SPLC),,1999.

[6] Binder, R. Testing object-oriented systems: models, patterns, and tools.Addison-Wesley Longman Publishing Co., Inc., 1999

[7] Ceccato, M., Tonella, P., e Ricca, F. Is aop code easier or harder to testthan OOP code? Workshop on Testing Aspect Oriented Programs, 2005.

[8] Colyer, A. Rashid, G. Blair, On the Separation of Concerns in Program Families. Technical Report COMP- 001-2004,

[9] Czarnecki, K., Eisenecker, U., Generative Programming: Methods, Tools, and Applications, Addison-Wesley, 2000.

[10] Filman, R., Elrad, T., Clarke, S., Aksit, M., Aspect-Oriented Software Development. Addison-Wesley, 2005.

[11] Kiczales, G., et al. Aspect-Oriented Programming. Proc. of ECOOP'97, 1997.

[12] Knauber, P. , Product Line Testing and Product Line Development - Variations on a Common Theme, Proceedings of International Workshop on Software Product Line Testing (SPLiT 2005)

[13] Kulesza, U., Alves, V., Garcia, A., Lucena, C., Borba, P. Improving Extensibility of Object-Oriented Frameworks with Aspect-Oriented Programming, Proceedings of ICSR'2006, June 2006.

[14] Kulesza, U., Coelho, R., Alves, V., Garcia, A., Lucena, C., Borba, P, Implementing Framework Crosscutting Extensions with EJPs and AspectJ, Proc. of Brazilian Symposium on Software Engineering , 2006 (to appear).

[15] Larochelle, D. et al., Join Point Encapsulation, Proc. Workshop Software Eng. Properties of Languages for Aspect Technologies, 2003.

[16] Lesiecki, N., AOP@Work: Unit test your aspects, developerWorks, November 2005.

[17] Lopes, C. and Ngo, T.. Unit testing aspectual behavior. In Proc. Workshop on Testing Aspect-Oriented Programs, 2005.

[18] Mackinnon, T., Freeman, S., and Craig, P. EndoTesting: Unit Testing with Mock Objects. Proc. of XP'2000, 2000.

[19] Mattson, M., Bosch, J., Fayad, M., Framework Integration: Problems, Causes, Solutions. Communications of the ACM, 42(10):80–87, 1999.

[20] McGregor, J.D., Testing a Software Product Line, Technical Report, CMU/SEI-2001-TR022.

[21] Mortensen, M., Ghosh, S. Using Aspects with Object-Oriented Frameworks, Proc. of AOSD'2006, Industry Track, 2006.

[22] Muccini, H.,and Hoek, A., Towards Testing Product Line Architectures, ETAPS2003 workshop.

[23] Myers, G. J. The Art of Software Testing. Wiley, 2nd Ed. 2004.

[24] Riehle, D., Gross, T. "Role Model Based Framework Design and Integration". Proceedings of OOPSLA'1998, pp. 117-133.

[25] Sullivan, K. et al. Information Hiding Interfaces for Aspect-Oriented Design, Proceedings of ESEC/FSE´2005, pp.166-175, 2005.

[26] Xie, T., Zhao, J., Marinov, D., and Notkin, D. Automated test generation for AspectJ programs, AOSD 2005 Workshop on Testing Aspect-Oriented Programs, Chicago, 2005.

[27] Zhou, Y., Richardson, D., and Ziv, H. Towards a practical approach to test aspect-oriented software. In Proc Workshop on Testing Component-Based Systems (TECOS), 2004.