

Instantiating and Customizing Product Line Architectures using Aspects and Crosscutting Feature Models

Uirá Kulesza¹ Alessandro Garcia² Fábio Bleasby¹ Carlos Lucena¹

¹*Software Engineering Laboratory – Computer Science Department
Pontifical Catholic University of Rio de Janeiro - PUC-Rio - Brazil
{uira, fabiobleasby, lucena}@inf.puc-rio.br*

²*Computing Department – InfoLab 21
Lancaster University – UK
garciaa@comp.lancs.ac.uk*

Abstract

The development of highly customizable software family architectures requires the explicit handling of crosscutting features through domain engineering and application engineering. This work explores the use of AOSD techniques in the customization and instantiation of product line architectures. Our method allows an improved customization and instantiation of frameworks by using crosscutting feature models. It also intends to provide guidelines to modularize the implementation of framework features using aspects. In this paper, we present an overview of our method by describing the main models that allows the implementation of aspect-oriented generative approaches. The method is illustrated by presenting its application in the refactoring of the JUnit framework.

1. Introduction

The development of highly customizable software family architectures is directly dependent on the conception of representation techniques and respective tools to support variability of crosscutting features. Aspect-Oriented Software Development (AOSD) [8] is an evolving paradigm that modularizes crosscutting concerns which existing paradigms are not able to capture explicitly. However, Aspect-Oriented (AO) techniques have mostly been explored to support a clean separation of crosscutting concerns through application development.

Only recent research work has explored the use of AO techniques in the development of software product-lines [1, 2, 12, 14, 16]. These works have demonstrated that AO techniques can bring benefits for the development of software product lines (or software families). Our experience have shown the following advantages [11, 12, 13, 14]: (i) clear separation of crosscutting features starting at early design phases; (ii) direct mapping of crosscutting features in aspects; (iii) simplified implementation of

code generators, because the composition of crosscutting features is accomplished by the aspect weavers, and (iv) improved reuse of artifacts associated with crosscutting features.

In previous work, we have presented an AO generative approach for the context of multi-agent system development [14], and notations to represent crosscutting relationships in feature models and software architecture models [4, 12, 13]. However, a number of important issues remain unaddressed, such as: How to express the variability of crosscutting features through out the domain engineering process? How to support the mapping of crosscutting features to architecture and implementation artifacts of a product line? How to modularize and implement the software product line architecture in order to make easier its customization to different products? How to support the automatic instantiation of product line architectures?

In this work, we explore the use of AO techniques in the customization and instantiation of software family architectures. Our studies concentrate mainly on the context of frameworks, a common technology used in the design and implementation of software product line architectures. We propose a generative method [6] that allows the customization and instantiation of frameworks by using crosscutting feature models. It also intends to provide guidelines to modularize the implementation of framework features using aspects. In this paper, we present an overview of our method by describing the main models that allow the implementation of aspect-oriented generative approaches. Our approach is illustrated with its application in the refactoring of the JUnit object-oriented framework.

The remainder of this paper is organized as follows. Section 2 gives a brief description of our approach. Section 3 presents the application of our method in the JUnit framework case study. We initially show the refactoring of the JUnit framework using aspect-oriented programming. After that, we describe the configuration of the JUnit framework to be

customized and instantiated based on a crosscutting feature model. Section 4 presents our conclusion and points to directions for future work.

2. Approach Overview

Our method focuses on the definition of a generative model similar to the presented by Czarnecki and Eisenecker [6]. However, we propose the extension of that generative model to enable the use of AO techniques in the implementation, instantiation and customization of AO architectures. The generative model of our method is composed of three models:

(I) a crosscutting feature model – which works as a configuration domain-specific language (DSL) responsible to specify and collect the features to be instantiated in the software family architecture. We are currently refining an existing meta-model [7] to represent crosscutting relationships between features. An example of a crosscutting feature model is presented in Section 3;

(II) an AO architecture model – which defines the main components of a software product line architecture. The architecture model is represented using two synchronized representations: (i) a model in aSideML [4, 5], an extension to the UML meta-model that allows the specification of aspects; and (ii) source-code in Java and AspectJ programming languages. The source-code represents the implementation of an AO product line architecture addressing the different variabilities. We also offer guidelines to implement AO architectures by means of a core OO framework, a set of AO software libraries that extend the base framework, and a set of aspects which define alternative compositions

between the framework and OO extensions. Each component (framework, aspect library or OO extension) is specified as a set of classes, aspects and templates. The latter ones define elements that will be customized during the instantiation of the architecture;

(III) a configuration model – which specifies the mapping between the features existing in the crosscutting feature model and the components (or their respective sub-elements, such as, class, aspect or templates) of the AO architecture. The configuration model is used to support the decision of which components must be instantiated and what customizations must be realized in those components considering a specific member of the product line (represented by an instance of a feature model).

Figure 1 illustrates the relationships between the different models existing in our method. It presents the models in the perspectives of domain implementation and application engineering [6]. In domain implementation, the feature, architecture and configuration models are specified considering a specific software product line. During the instantiation of a member of the product line (application engineering), an instance of a feature model is defined by the application engineer. A tool uses this model to customize and instantiate the AO architecture defined for that domain. This process is also based on the configuration model used, which defines the mapping between features and architecture components. Our tool is being developed as an Eclipse [18] plugin detailed in Section 3.4.

The next section describes and gives examples of these models in the context of the JUnit framework.

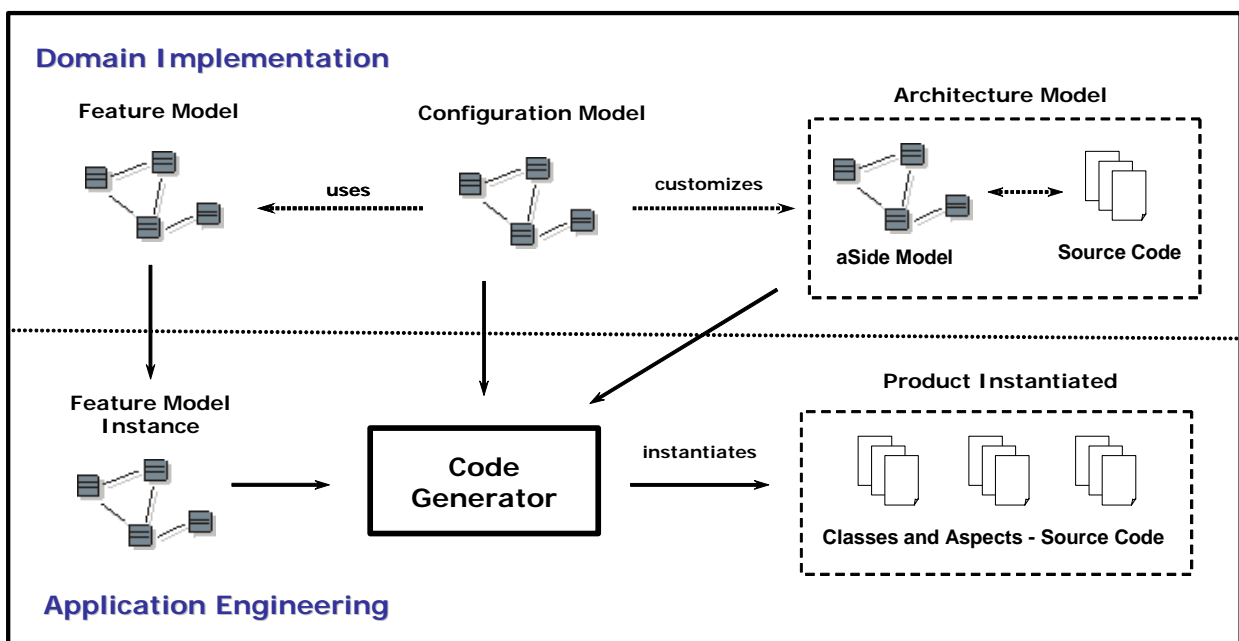


Figure 1. Approach Overview

We show how the crosscutting feature models and aspects can be used together to improve the customization and instantiation of the JUnit architecture.

3. The JUnit Case Study

This section presents the refactoring of the JUnit¹ framework using our approach.

3.1. The JUnit OO Implementation

The JUnit framework implementation is composed of the following components:

- (i) **Tester**: this component defines the core framework classes which are responsible to specify the basic behavior to execute test cases and suites. The main hot-spot classes available in this component are `TestCase` and `TestSuite`. The framework users extend these classes in order to create specific test cases to their applications;
- (ii) **Runner**: this component is responsible for offering an interface to start and track the execution of test cases and suites. JUnit provides three alternative implementations of test runners, as follows: a command-line based user interface (UI), an AWT based UI, and a Java Swing based UI;
- (iii) **Extensions**: this component is responsible to define classes which extend the basic behavior of the JUnit framework. Examples of available extensions are: a test suite to execute tests in separate threads, a test decorator to run tests repeatedly, and a test setup class that allows to specify initial and final configuration to specific tests.

The JUnit framework has been implemented using classical design patterns, such as, Observer and Decorator [9]. However, the use of these patterns brings difficulties to the framework understanding, configurability, and maintainability, particularly with respect to the interactions between the main JUnit components. The Observer pattern is used to notify external components (test runners, for example) about the status of test cases execution. The Observer-relative code is tangled with the code of the `TestResult` class. This design decision brings difficulties to understand the core behavior of the Tester component, as well as, the composition code between the Tester and Runner components. Moreover, if a software developer intends to monitor other internal events of interest in the framework new invasive changes need to be accomplished.

Also, many of the classes in the JUnit's Extensions component are implemented using OO inheritance mechanisms. This design structuring makes it difficult to understand how those extension classes are related to internal Tester framework classes. Moreover, it also restricts the extension possibilities you can define to the Tester component classes, and

requires the JUnit users to understand the new extension classes when instantiating the framework.

3.2. AO Refactoring of JUnit Implementation

The tangling-related problems of the JUnit framework (Section 3.1) were analyzed and different alternatives of AO refactorings were evaluated. The activities of the JUnit refactoring were mainly supported by the catalog of AO refactorings presented in [17]. Figure 2 depicts the resulting JUnit AO design. The composition between the Tester and Runner components which was specified by the OO implementation of the Observer pattern was replaced by a set of "observer aspects". These aspects are responsible to notify runners of the status of test cases execution. This design decision brings a better separation of the composition code from the testing and runner components.

The Extensions component was also implemented as a set of aspects. These aspects are used to extend specific framework classes (such as, `TestCase` and `TestSuite` mentioned in Section 3.1). They introduce crosscutting behavior related to the execution of test cases and suites. This design decision avoids the use of inheritance mechanisms which brings more complexity to the understanding of the OO JUnit design. With the use of aspects, the framework users do not need to understand or define subclasses of those specific extensions. Besides, as we will see in Section 3.5, using generative techniques we can configure those extensions to be applied to specific sets of test cases and suites, thereby increasing the JUnit configurability.

In order to enable the instantiation and customization of the JUnit framework, our approach defines additional templates which represent the framework hot-spots instances. These instances will be created and customized based on a feature model instance. Templates in our approach are used to specify elements (classes, aspects, files) that will be customized during the instantiation of the architecture. Figure 2 presents several templates, such as:

- (i) `TestCaseTemplate` and `TestSuiteTemplate` which are used to create respectively specific test case and test suite classes for a given application;
- (ii) `ObserverTestTemplate` – defines which GUI alternative will be plugged in the Tester component to monitor the execution of test cases;
- (iii) `RepeatTestTemplate`, `ActiveTestTemplate` and `TestSuiteDecoratorTemplate` – which are customized to extend test cases and suites with additional functionality.

3.3. The JUnit Crosscutting Feature Model

The JUnit crosscutting feature model expresses the framework variabilities by exploiting aspect-oriented

abstractions. Figure 3 depicts the resulting feature model. For simplicity purposes, we have omitted some irrelevant details in the model. It is composed of three main features (testing, runner, extensions) which are directly mapped to JUnit components. We extend the feature model proposed in [7], which also allows the representation of feature cardinality.

The testing and runner features do not crosscut other features and, as a result, are modeled as typical features in a feature model. The testing feature is composed of several test suite features. Each test suite feature aggregates its respective test cases. Both test suite and test case features are used to instantiate JUnit framework by defining the automated tests for a specific system. The runner feature models the alternative GUI options available.

The extensions feature specifies the different extensions that can be used with test cases and suites. The extensions features are all examples of *crosscutting features*, since they can be directly applied to extend the test case and test suite features. In our model, crosscutting features are used to model features which extend other features; they are composed of *pointcut features* which can traverse and extend *joinpoint features*. Joinpoint features can be extended by specific crosscutting features, i.e. by means of the pointcut features.

The configuration model (Section 3.4) restricts which joinpoint features can be traversed by which pointcut features. In other words, this model specifies the crosscutting possibilities with respect to a pair of features. Figure 3 depicts the following crosscutting features: repeat, concurrency, and test setup. Each of them has associated pointcut features, which are used to define a `<<crosscuts>>`

relationship with joinpoint features. Currently, we are using the existing “reference” relationship defined in [7], to represent these `<<crosscuts>>` relationships.

Our approach allows the creation of `<<crosscuts>>` relationships both in feature models representing product lines and in feature model instances (a feature configuration in [7]) representing applications. This strategy improves the flexibility in the framework customization. For example, in the JUnit case study, we can define in a feature model that we will apply the concurrency feature to every test case of the system. Figure 3 illustrates this situation: every test case that we instantiate using that feature model will be executed in a different thread. Alternatively, we can postpone the use of `<<crosscuts>>` relationship to the specification of the feature model instance. Figure 4 presents, for example, that only two specific test cases being instantiated will be executed repeatedly.

3.4. The Configuration Model

The configuration model is used to specify the configuration knowledge [6] necessary to customize and instantiate AO product line architectures using feature models. It is composed of:

- (i) description of *dependency relationships* between the architecture model’s components (and sub-elements) and the features specified in the feature model;
- (ii) definition of *valid crosscutting relationships* between pointcut and joinpoint features; and

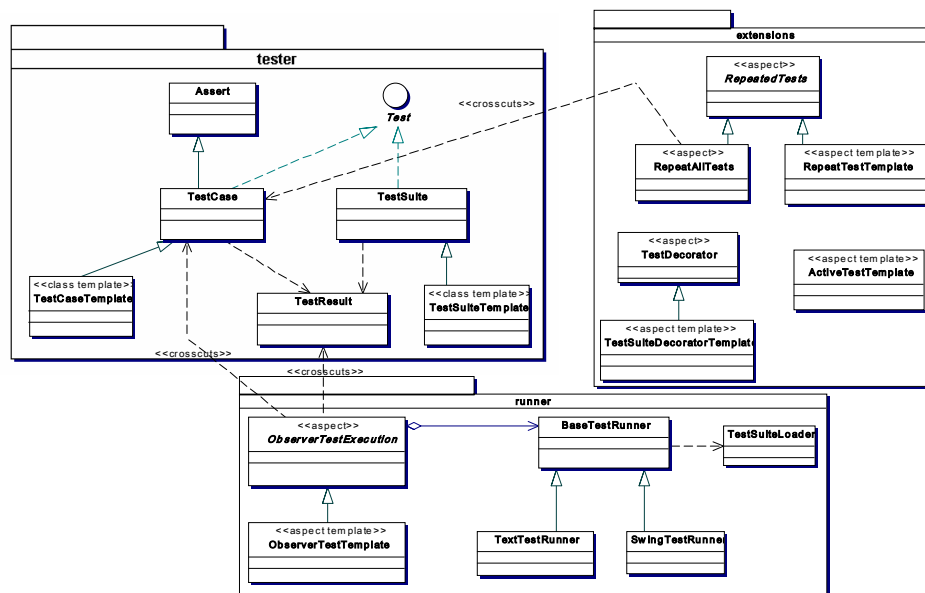


Figure 2. AO Refactoring of the JUnit Implementation

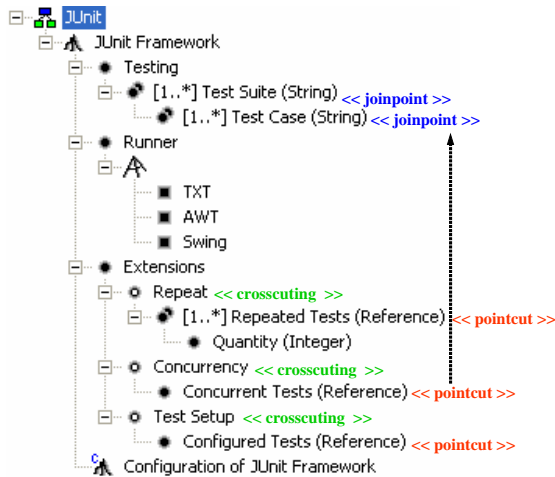


Figure 3. JUnit Feature Model

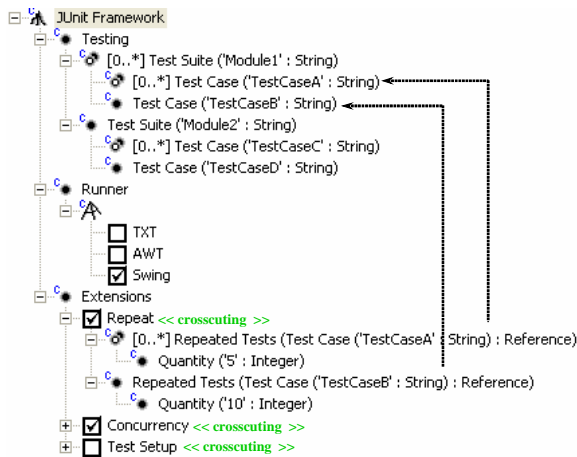


Figure 4. JUnit Feature Model Instance

(iii) mapping between joinpoint features and specific join points in classes of the AO product line architecture.

Figure 5 shows examples of these configuration issues in the context of the JUnit framework. Although the figure only shows configuration options in a textual representation, we are also currently working in visual representations to them.

The dependency relationships between architecture components (and sub-elements, such as, classes, aspects and templates) and features allow to specify which components must be instantiated when specific features are selected. When defining the dependency relationships, the domain engineers use the following guidelines: (i) if a component (or sub-element) must be instantiated to every product of the product line, then no dependency relationships need to be specified; (ii) if a component (or sub-element) depends on the occurrence of a specific feature, a dependency relationship must be created between them.

These dependency relationships are used by our code generator to decide which components, classes and aspects will be included in a product based on feature model instances defined by application engineers. In case of templates, the dependency relationships define if they will be processed and included in the final product generated. Figure 5 shows, for example, that the runner classes (*TextTestRunner*, *SwingTestRunner*, etc) will be created only if the respective associated features were also selected in the feature model instance. We can also observe that all the template elements depend on specific features which provide knowledge necessary for their instantiation.

The configuration model also defines potential relationships between pointcut and joinpoint features. This information allows the code generator to check if the application engineers have specified valid crosscutting relationships. Figure 5 shows the valid crosscutting relationships for the JUnit example, which specify that all the extension features can crosscut the test case and suite features.

Finally, the configuration model must also define the mapping between joinpoint features and specific join points in classes of the product-line. This mapping is useful during application engineering to make the customization of pointcuts easier in aspect libraries; it uses only information provided by a feature model instance. The mapping involves the identification of which parts of classes (e.g.: constructor execution and method call) correspond to specific joinpoint features. The mapping refers to specific and valid AspectJ join points. The JUnit configuration model (Figure 5) presents the mapping of test case and suite features to specific AspectJ join points. These AspectJ join points represent, respectively, the *run()* method execution in *TestCase* and *TestSuite* classes. Currently, we are also analyzing the possibility of using Java annotations to embed the joinpoint feature directly in the source code¹.

3.5. The Customization and Instantiation Process

Our approach supports the application engineering process through a tool which instantiates and customizes AO software family architecture. The application developer provides a specific configuration model and a feature model instance. Our tool is being developed as a set of Eclipse plugins. In a previous work, we have already presented a previous version of our prototype tool [14].

We are currently integrating the feature model plugin (*fmp*) [3] to address both the modeling of feature model and feature model instances. As previously

¹ We are working on the Java annotation alternative, because AspectJ 5.0 already allows to specify pointcuts using annotations.

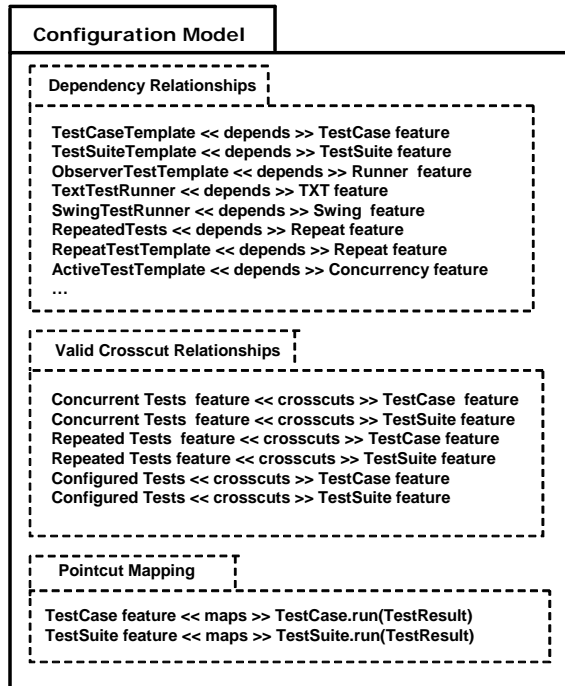


Figure 5. JUnit Configuration Model

mentioned, we are extending the feature meta-model of *fmp* to support the representation of crosscutting features.

An AO architecture model in our tool is specified using a synchronized representation of aSideML models and Java and AspectJ source code. The implementation of aSideML models is being realized using the following technologies provided by Eclipse platform: EMF, GEF and UML2.

The configuration model is being implemented as a set of Eclipse views which allow to specify: (i) the dependency relationships between features in *fmp* models and elements (classes, interfaces, aspects, templates, methods, etc) of aSideML models; (ii) the valid crosscutting relationships that can exist between pointcut and joinpoint features in *fmp* models; and (iii) the mapping between joinpoint features in *fmp* models and join point in Java classes.

The code generator of our tool is triggered by the application engineers. The generator uses the three models (feature model instance, architecture and configuration models) to customize and instantiate the AO architecture. During the instantiation process, the code generator realizes a sequence of three main steps. First, it verifies if there is any invalid crosscutting relationship between features. The detection of such a problem interrupts the process, and the application engineer is in charge of solving the conflict.

Second, the architecture model is traversed and it proceeds as follows. For each component encountered the code generator verifies in the configuration model if it depends on any special

feature. In such case, the code generator only instantiates that component (and processes respective sub-elements) if there is an occurrence of that special feature in the feature model instance. The components are instantiated by creating a correspondent Java package. When processing sub-elements of a component (classes, interfaces, aspects, templates, methods, etc) the same process is applied. It means that it is also verified if the sub-element depends on any feature as a condition to instantiate it. The template elements must always depend on any feature. Therefore, they are processed for every occurrence of that feature. During the template processing the information about the feature (and respective sub-features) which it depends is used to support the template customization.

Third, the crosscutting relationships specified in feature models are used by the code generator to customize which aspects must affect which classes of the system. Templates of aspects are processed during this process. It represents an aspect that will be customized. In Figure 5, the `ObserverTestTemplate`, `RepeatTestTemplate`, `TestSuiteDecoratorTemplate`, `ActiveTestTemplate` templates represent aspects that must be customized. Every template of aspect depends on a crosscutting feature. So, when it is processed it uses information from this crosscutting feature, such as, its pointcut features and respective joinpoint features. A template of an aspect can have its pointcuts customized based on information from the joinpoint features. The mapping between join point features and join points in classes specified in the configuration model is used in this process.

Consider, for example, the JUnit customization and instantiation using the feature model instance, the configuration model and the AO architecture depicted, respectively, in Figures 4, 5 and 2. When these models are processed by the code generator, JUnit is instantiated by:

- (i) creating one test suite (`Module1` and `Module2`) and four test case (`A`, `B`, `C` and `D`) classes by processing the `TestCaseTemplate` and `TestSuiteTemplate` templates;
- (ii) including the classes of the swing feature and discarding the classes that implement the `awt` and `txt` features. Also, the aspect template `ObserverTestTemplate` is customized to reference the classes of the swing feature.;
- (iii) creating two aspects that implement the repeat feature for two test case classes by means of processing the `RepeatTestTemplate`. The pointcuts of these aspects are customized using information from the configuration model;
- (i) discarding the aspects that implement the `Test Setup` extension feature;

(v) and finally, creating an aspect which advises all the test case classes to execute them in separate threads. The `ActiveTestTemplate` is used in this process.

4. Conclusions and Future Work

This paper presented our ongoing work on the definition of a systematic method to develop aspect-oriented generative approaches. We also described the main models in our approach that allows the implementation of aspect-oriented generative approaches. Our method proposes the use of crosscutting feature models in order to: (i) deal with the specification of crosscutting relationships between features and (ii) allow improved customization and instantiation of product lines. The approach was illustrated by presenting its application in the JUnit framework.

In a previous work [14], we implemented an aspect-oriented generative approach for the development of multi-agent systems. In that work, we have observed several benefits on the use of AO techniques to develop generative approaches, such as [12, 13, 14]: (i) clear separation of crosscutting features at early development phases; (ii) direct mapping of crosscutting features in aspects; and (iii) simplification of the code generators.

This paper refines our previous work by extending feature models with new AO abstractions to support improved customizability of product lines. In addition, we present a configuration model definition, which allows to specify separately the mapping between features and components and enables the customization of pointcuts from the feature models.

We are currently implementing a tool to support all the models presented in the paper. Also, new case studies involving different software families are being realized to validate our approach. In these case studies, we also intend to explore the instantiation of aspect libraries using our approach. Finally, a set of guidelines to modularize the implementation of framework features using aspects is also being refined.

Acknowledgements. We would like to thank Vander Alves and Itana Gimenes for fruitful discussions on the ideas developed in this paper. This work has been partially supported by FAPERJ under grant No. E-26/151.493/2005 and by CNPq under grant No. 140252/2003-7 for Uirá. The authors are also supported by ESSMA under grant 552068/2002-0 and by European Commission grant IST-2-004349: European Network of Excellence on Aspect-Oriented Software Development (AOSD-Europe), 2004-2008.

References

- [1] V. Alves, et al. "Extracting and Evolving Mobile Games Product Lines". In 9th International Software Product Line Conference (SPLC'05), September 2005 (to appear).
- [2] M. Anastasopoulos, D. Muthig. "An evaluation of aspect-oriented programming as a product line implementation technology". In Proceedings of the International Conference on Software Reuse, 2004.
- [3] M. Antkiewicz and K. Czarnecki. FeaturePlugin: Feature modeling plug-in for Eclipse, OOPSLA'04 Eclipse Technology eXchange (ETX) Workshop, 2004.
- [4] C. Chavez, A. Garcia, U. Kulesza, C. Sant'Anna, C. Lucena. "Taming Heterogeneous Aspects with Crosscutting Interfaces". Proceedings of the Brazilian Symposium on Software Engineering (SBES'2005), October 2005. (to appear)
- [5] C. Chavez. "A Model-Driven Approach to Aspect-Oriented Design". PhD Thesis, PUC-Rio, April 2004.
- [6] K. Czarnecki, U. Eisenecker. Generative Programming: Methods, Tools, and Applications, Addison-Wesley, 2000.
- [7] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration using feature models. In Proceedings of the Third Software Product-Line Conference, September 2004.
- [8] R. Filman, T. Elrad, S. Clarke, M. Aksit. Aspect-Oriented Software Development, Addison-Wesley, Boston, 2005.
- [9] E. Gamma, et al. "Design Patterns: Elements of Reusable Object-Oriented Software". Addison-Wesley, Reading, 1995.
- [10] G. Kiczales, et al. "Aspect-Oriented Programming". Proc. Of ECOOP'97, LNCS 1241, Springer, Finland, June 1997.
- [11] U. Kulesza, A. Garcia, C. Lucena. "Generating Aspect-Oriented Agent Architectures". Proc. 3rd Workshop on Early Aspects, 3rd International Conference on Aspect-Oriented Software Development, Lancaster, UK, March 2004.
- [12] U. Kulesza, A. Garcia, C. Lucena, A. Von Staa. "Integrating Generative and Aspect-Oriented Technologies". Proceedings of the Brazilian Symposium on Software Engineering, pp. , Brazil, October 2004.
- [13] U. Kulesza, A. Garcia, C. Lucena. "Towards a Method for the Development of Aspect-Oriented Generative Approaches". Proceedings of the 4th Workshop on Early Aspects, OOPSLA'2004, October 2004, Vancouver.
- [14] U. Kulesza, A. Garcia, C. Lucena, P. Alencar. "A Generative Approach for Multi-Agent System Development". In "Software Engineering for Multi-Agent Systems III". Springer-Verlag, LNCS 3390, pp. 52-69, December 2004.
- [15] U. Kulesza, C. Sant'Anna, C. Lucena. "Refactoring the JUnit Framework using Aspect-Oriented Programming", Poster Session, OOPSLA'2005, October 2005, San Diego (to appear)..
- [16] M. Mezini, K. Ostermann: "Variability management with feature-oriented programming and aspects". Proceedings of Foundation on Software Engineering (FSE'2004), SIGSOFT, pp. 127-136, 2004.

- [17] M. Monteiro, J. Fernandes. Towards a Catalog of Aspect-Oriented Refactorings. In Proceedings of the AOSD'05, Chicago, March 2005.
- [18] S. Shavor, J. D'Anjou, S. Fairbrother, et al. The Java Developer's Guide to Eclipse. Addison-Wesley, 2003.