

# ULRR

## Feature-modeling and aspect-oriented programming: integration and automation.

Item Type	Meetings and Proceedings
Authors	Lee, Kwanwoo;Botterweck, Goetz;Thiel, Steffen
Citation	10th ACIS International Conference on Software Engineering;2009
Publisher	IEEE Computer Society
Download date	2026-06-12 23:22:07
Item License	<a href="https://creativecommons.org/licenses/by-nc-sa/1.0/">https://creativecommons.org/licenses/by-nc-sa/1.0/</a>
Link to Item	<a href="https://hdl.handle.net/10344/1850">https://hdl.handle.net/10344/1850</a>

# Feature-Modeling and Aspect-Oriented Programming: Integration and Automation

Kwanwoo Lee\*, Goetz Botterweck<sup>†</sup> and Steffen Thiel<sup>‡</sup>

*\*Department of Information Systems Engineering  
Hansung University, Seoul, South Korea  
Email: kwlee@hansung.ac.kr*

*<sup>†</sup>Lero – the Irish Software Engineering Research Centre  
University of Limerick, Limerick, Ireland  
Email: goetz.botterweck@lero.ie*

*<sup>‡</sup>Department of Computer Science  
Furtwangen University of Applied Sciences, Furtwangen, Germany  
Email: steffen.thiel@hs-furtwangen.de*

## Abstract

*Feature modeling is an essential activity for modeling and managing the variability of a software product line. On the other hand, aspect-oriented programming provides effective means for modularizing feature implementation. Although current AOP tools (e.g., AJDT) provide a mechanism for switching aspect modules on and off to configure a product, this becomes infeasible in the context of large-scale product lines with thousands of variations. In this paper, we describe how feature modeling can be integrated with aspect-oriented programming to perform automated product derivation efficiently and effectively in the context of large-scale product lines.*

## 1. Introduction

A software product line (SPL) is a set of software intensive systems that share a common, managed set of features and are developed from a common set of core assets in a prescribed way [1]. In software product line engineering (SPLE), products are derived based on product configuration by selecting, adapting, and configuring parts of core assets for the SPL.

Feature modeling provides effective means for managing and representing the product configurations of a SPL in terms of feature configurations. In this paper, we assume that the products of a SPL are created based on configurations of a feature model. To be able to use these feature configurations to derive products from the core assets, these assets must be designed in a way that allows to switch variable parts on and off depending on the selection of variable features in the product configuration. However, a feature does not always correspond to exactly one implementation component.

Aspect-oriented programming (AOP) [2] is a good candidate for modularizing feature implementation, as it provides effective mechanisms for encapsulating crosscutting concerns into modular units. There have been several attempts to modularize features using aspects [3], [4], [5].

Although AOP development tools, such as the AspectJ development tools (AJDT), provide a mechanism for setting up different configurations of components (with some components selected and some deselected), with an increasing size of the product line, it simply becomes infeasible to manage features and their corresponding components manually.

To address the complexity and scalability challenges, we propose a model-driven approach to automating product derivation by integrating feature modeling and aspect-oriented programming:

*Model-based description of a SPL:* In the approach, we describe a SPL with three models, i.e., a feature model, an implementation model, and mappings between them. The feature model describes configurable options for products in the SPL, while the implementation model is the abstract representation of program code within the SPL. The implementation model allows to associate feature model elements with program elements.

*Automated product derivation:* We use a model-transformation to describe the process that takes a feature configuration (i.e., a set of features selected for a particular product) as input and produces an assembled and executable software product as output. The derivation mechanisms are implemented partially by declarative model transformations, and partially by the AspectJ development tools.

For a better understanding, the next section presents the background and the overview of our approach. Section 3 describes the three models used to setup a software product line. Section 4 presents how these models are used in model-driven product derivation processes to create an executable product. Our approach is compared to related work in

section 5. Finally, we conclude the paper with a discussion of the presented approach.

## 2. Background

### 2.1. Feature Modeling and Configuration

A *Feature Model*, which is the output of feature modeling, describes configuration choices for the products of a SPL in the form of an AND/OR tree. *Mandatory* features have to be included in all product configurations, if all of their ancestors in the tree are included in the configurations. *Optional* features represent choices to choose from, depending on requirements for a particular product. In addition, *group* features, such as XOR (OR) features, describe the constraints that exactly one (at least one) of the features must be included in a product configuration. Moreover, configuration dependencies between features (e.g., *configuration inclusion* and *configuration exclusion*) further constrain the configuration choices of a feature model. More details on feature modeling can be found in [6], [7].

A feature configuration describes the features selected for a particular product. The features in a feature configuration must not violate any constraints or dependencies specified in the feature model. Numerous tools have been developed, for instance BigLever Gears [8], Feature Model Plug-in [9], Pure::variants [10], to support the process of feature configuration.

### 2.2. Aspect-Oriented Feature Implementation and Configuration

In general, a feature does not always correspond to exactly one implementation component. Therefore, there may be complex mapping relationships between features and their implementation. If a feature is related to several parts of multiple components, it becomes very difficult to select and compose corresponding code fragments according to a selected feature configuration. A first step toward flexible feature composition are approaches that allow to assemble an implementation based on modularized units of composition.

AOP provides an effective way of incorporating variable features into a product, as it can isolate variable features into aspects and integrate them in an additive way. That is, based on an initial base structure implementing common features, variable features can be implemented using aspects. Then, an aspect weaver creates products by weaving aspects (implementing variable features) into the base modular structure.

Unfortunately, AOP languages (e.g., AspectJ) provide no mechanism to link features with its program elements. The only way to compose a product with features is to manipulate the compiler's build path or to manually include or exclude certain aspects which implement the selected

features. This makes product derivation difficult, error-prone, and time consuming in the context of a large-scale product line, which is the main motivation behind the research discussed here.

### 2.3. Overview of Automating Feature-Oriented Product Derivation

In our approach the software product line is described in terms of three models or source code artifacts respectively (see the markers **A** to **D** in Figure 1).

During Feature Modeling **1**, a Feature Model **A** is created, which describes the capabilities of the product line including configuration options for products.

During Implementation Structuring **2**, implementation modules must be constructed and organized in a way that they can be effectively configured based on the feature configurations of a SPL.

The Implementation Model **B** provides an abstracted view on this implementation. Just like the corresponding textual code, the implementation model contains concepts like *Aspect* or *Class* but describes them as model elements and contains references to the corresponding constructs in the textual programming languages. This description of the implementation in the form of an abstracted model allow us to describe mappings **B** between features and the corresponding code components (e.g., aspects or classes).

During Feature Implementation **3** the capabilities of the software product line are realized with technologies like aspect-oriented programming (e.g., AspectJ) or object-oriented programming (e.g., Java).

It should be noted that in this paper we focus on the processes and mechanisms for automating product derivation. More details on the mentioned processes (**1** to **3**), which are part of the Domain Engineering activities, can be found in [11], [12].

With the described product line artifacts (**A** to **D**) as input we can start the product derivation processes (**4** to **6**) which will derive the assembled executable product. Before doing so, we will now turn our attention to the models and modeling languages, which are used to describe the product line and are required as input for the derivation process.

## 3. Modeling Feature-Implementation Mapping

Because we have captured the implementation not only in programming languages, but also as a corresponding implementation model, we can now describe mappings **B** between features and their implementation. The mappings enable us to select corresponding implementation units based on a feature configuration. This is used during the derivation **4** of the Implementation Configuration.

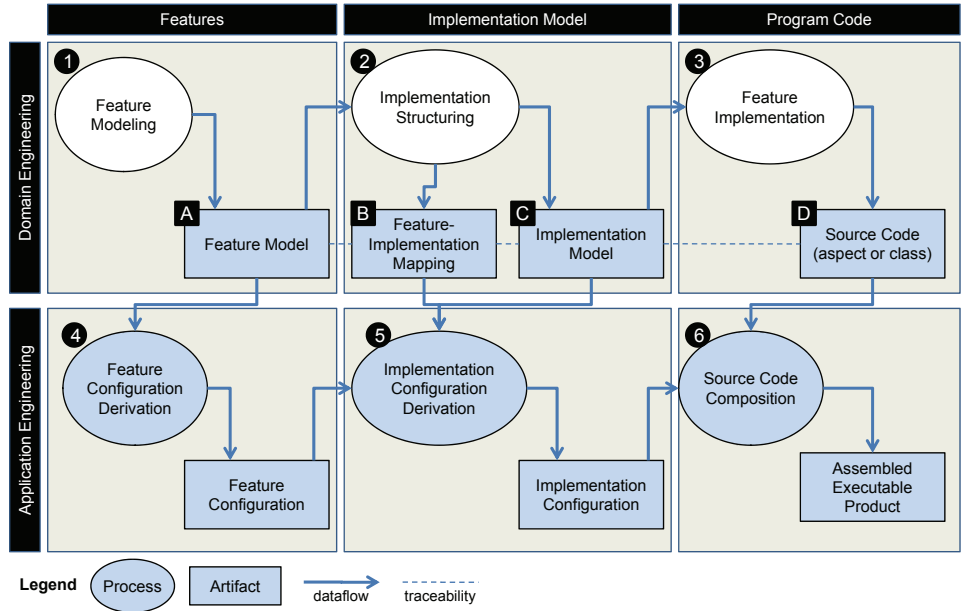


Figure 1. Overview of the presented approach.

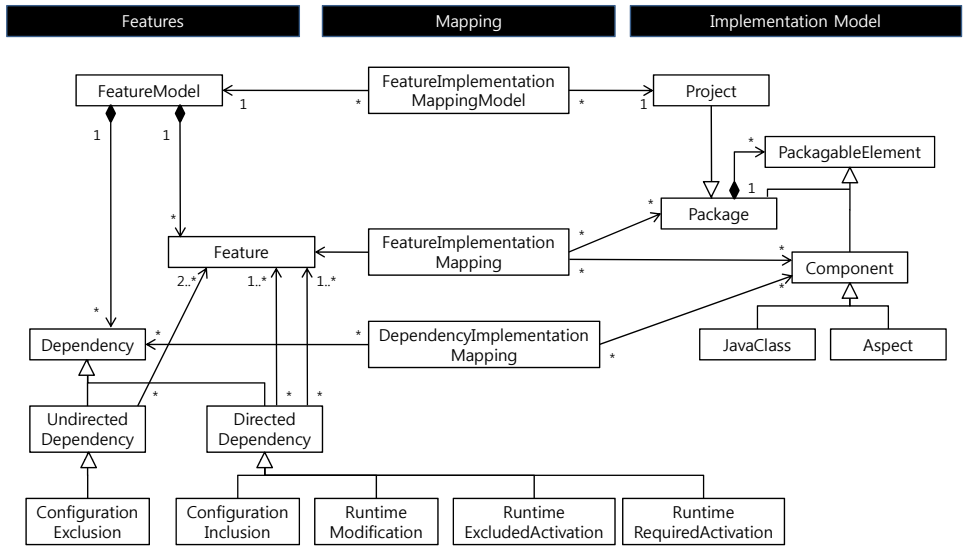


Figure 2. Meta-models and links between them: Features, Mapping (FIM) and Implementation model (AML).

Basically such a model describes mappings between features and the corresponding components, which can be either object-oriented (Java Classes) or aspectual components (AspectJ Aspects). To be able to process this knowledge in a model-driven approach we have to capture it in a precisely defined language. Hence, we define the meta-model for the feature model, the implementation model, and their mappings (See Figure 2).

The feature meta-model (left part of Figure 2) defines concepts like Model, Feature, and various types

of dependencies, e.g., ConfigurationExclusion or RuntimeExcludedActivation.

The main structure of the implementation meta-model (right part of Figure 2, corresponding to 3 in the overview) is given by a Composite pattern: Packages contain PackagableElements which can be either other Packages or Components (Java Classes or Aspects).

Features and Dependencies can be related to Components by FeatureImplementation-

Mappings and DependencyImplementation-Mappings, respectively. These mappings are mostly one-to-one, however, in general we can describe n-to-m mappings. So we are able to express something like “whenever you select these  $n$  features, then you need these  $m$  components”. In propositional logic this can be represented as  $(f_1 \wedge \dots \wedge f_n) \Rightarrow (c_1 \wedge c_2 \wedge \dots \wedge c_m)$ . In addition to implication ( $\Rightarrow$ ) there are other possible interpretations of such references between features and their implementation. For more details, please refer to the discussion in [13].

This meta-model is partly based on our earlier work [14], which in turn is derived from [15].

## 4. Automated Mechanisms for Product Derivation

This section describes how the knowledge captured in models and code artefacts (**A** to **D**) is used during product derivation to create the assembled executable product.

Given the product-specific requirements, the first step performed during the creation of a product is the process of Feature Configuration Derivation **4**. Since this involves interpreting requirements, this activity cannot be automated and has to be performed by a software engineer. However, the process can be supported by interactive tools which provide visual guidance, for instance by highlighting parts of the feature model, which still require attention [14].

The feature configuration created during that process is used by the automated product derivation mechanisms **5** to **6**, which we explain in this section.

### 4.1. Derivation of Implementation Configuration

After the feature model has been configured, the next step is the Implementation Configuration Derivation **4** which reads this feature configuration, the model of the implementation, and the feature-implementation-mappings. From this input it determines an *Implementation Configuration* model which describes all components (classes and aspects) that have to be included to implement this particular feature configuration.

In our research prototype, this process has been implemented as a model transformation described in the Atlas Transformation Language (ATL). The corresponding ATL source code (an extract of the full model transformation) is listed below:

```

1 -- @atlcompiler atl2006
2 module DeriveModel_aml;
3
4 create amlOut : AMLMETA
5 from amlIn:AMLMETA, featureIn:FEATUREMETA,
6     fimIn:FIMMETA;
7 -- -----
8 -- Helpers
9 -- -----

```

```

10
11 -- select those mappings where *ALL* related
12 -- features are selected.
13 helper def : selectedFeatureImplementationMapping
14   : Sequence(FIMMETA!FeatureImplementationMapping)
15   = thisModule.allFeatureImplementationMappings
16     -> select( fim |
17               fim.features -> forall( feat |
18                                     feat.isSelected()
19               );
20 -- collect all components which are referenced
21 -- from selected mappings
22 helper def : selectedComponents
23   : Sequence(AMLMETA!Component) =
24   thisModule.selectedFeatureImplementationMapping
25     -> collect( fim | fim.components )
26     -> flatten();
27 [...]
28
29 -- -----
30 -- Definition of visibility
31 -- -----
32
33 helper context AMLMETA!Component def : isVisible()
34   : Boolean = thisModule.selectedComponents
35     -> includes(self);
36 [...]
37
38 -- -----
39 -- Copy Rules
40 -- -----
41
42 [...]
43 rule Aspect {
44   from s : AMLMETA!Aspect (
45     s.isVisible()
46   )
47   to t : AMLMETA!Aspect (
48     name <- s.name.debug('Aspect'),
49     id <- s.id
50   )
51 }
52
53 rule JavaClass {
54   from s : AMLMETA!JavaClass (
55     s.isVisible()
56   )
57   to t : AMLMETA!JavaClass (
58     name <. s.name.debug('JavaClass'),
59     id <. s.id
60   )
61 }
62
63 [...]

```

Three helper functions describe how the mappings are interpreted. `selectedFeatureImplementationMapping()` (lines 13-19) defines that a mapping is selected, if all related features are selected. `selectedComponents()` (lines 22-26) collects all components (i.e., aspects and classes) which are referenced by a selected mapping. `Component.isVisible()` (lines 33-35) defines that a component will be included during the derivation process, if it is one of these “selectedComponents”.

Based on these definitions, the copy rules (lines 42-63) perform the derivation by selectively copying elements from the product line implementation model. Since the resulting

model only contains those implementation elements which correspond to the requested feature configuration, this model represents the product-specific implementation configuration.

Subsequently the implementation configuration model is processed by a model-to-text transformation which generates the product-specific configuration in form of a textual file. This corresponds directly to the content given by the Implementation Configuration model. However, it is written in a simple textual syntax that can be directly processed by the AspectJ Development Tools. This model-to-text transformation has been implemented with Java Emitter Templates (JET).

## 4.2. Code Assembly

With the configuration files generated in the proceeding step we can now execute the assembly of the particular product ⑤.

In our research prototype this is realized by applying the Aspect Weaver of AspectJ. We use naming conventions and folder structures to organize the product configurations and corresponding generated products.

## 4.3. Orchestration of the Tool Chain

When we come up with the overall approach (see Figure 1), our goal was to automate the product derivation as far as possible. To go from product-specific requirements to an assembled executable product we have to perform several processes.

Within that context, only the feature configuration ④ has to be performed in an *interactive* fashion. The rest of the processes (⑤ to ⑥) can be executed mechanically.

To fully automate the approach we use Ant scripts, which orchestrate the overall process and custom Ant tasks, which integrate the various tools into this tool chain.

In summary, this tool chain turns a product-specific feature configuration into an *Assembled Application Implementation*.

## 5. Related Work

Aspect-oriented programming (AOP) techniques and languages were originally developed to modularize crosscutting concerns that would otherwise be scattered across multiple modular components. Recently, there have been several efforts to use one of them for modularizing feature implementation. Originally, Griss [16] proposed a conceptual framework of feature-driven, aspect-oriented product line engineering. Alves et al. [17] applied AOP in the development of mobile game product lines. Kästner et al. [4] implemented features of refactored Berkley DB using AspectJ.

AOP is not the only option that can be taken as a technique for modularizing feature implementation. Feature-oriented programming (FOP) [18] is an alternative to implement features. FOP takes features as first-class design and implementation entities. That is, features are designed and implemented as the refinements of an existing program and composed to form a complete system. Other programming techniques, such as Caesar [19], Framed Aspect [20], can be used for feature implementation.

These approaches mostly use one-to-one mapping (and do not support more complex mappings) between features and implementation units (e.g., aspects). This may not be a feasible solution, as features are not usually independent entities. Recently, Lee et al. [21] identified problems with the simple mapping between features and aspects and proposed detailed guidelines on how feature dependency information can be used for implementing features using AOP. The identified guidelines are utilized in this paper. However, the presented approach differs from [21] in that a systematic product derivation process is introduced.

Our approach deals with product derivation based on a feature configuration. The work in [22] used a template based approach for mapping features in a feature model to different kinds of models and a model-to-model transformation to instantiate models based on a feature configuration. However, in [22] the rules for filtering are described as OCL constraints, whereas we describe them as a model transformation which partly can be derived from the underlying meta-model. The work in [23] is similar to our approach in that domain artifacts are expressed using models, which are transformed from one model to another. AOP is used to implement crosscutting features on code level. However, the links between features and aspects are not made explicit.

In contrast to commercial tools such as pure::variants [10] and Gears [8] our approach uses models which are handled via the open source Eclipse Modeling Framework (EMF). Consequently, this provides the platform to apply techniques such as model transformations (e.g., ATL), graphical model editors (e.g., GMF) or other technologies from the Eclipse Modeling Project [24].

## 6. Conclusions

Our primary goal for the presented research is the automation of product derivation in a product line context. We have presented an approach to integrate feature modeling and AOP to facilitate a model-driven product derivation process, which turns a feature configuration into a fully assembled and executable product. This process is implemented as a research prototype, which automatically performs the necessary steps.

Although we have used AspectJ and its development environments (AJDT) to demonstrate the applicability of the proposed method, the method is not limited to specific

aspect-oriented programming languages. Since the mappings are defined in the meta language level, the method can be easily extended to support the derivation of products implemented with different programming languages (e.g., AspectC++, AspectJ, Java, C++, etc.).

The description of the software product line as a set of models allows to apply automated analyses techniques [25]. For instance, we could check an Java/AspectJ implementation for conformance with the design specified in the implementation model. Or we could extract the implementation model from the Java/AspectJ source code and then check the consistency between the implementation model and the feature model. For instance, when there is a dependency discovered in the implementation, which has not yet been reflected in the corresponding features [13].

## Acknowledgment

This work was supported, in part, by the Korea Research Foundation Grant funded by the Korean Government (MOEHRD, Basic Research Promotion Fund) (KRF-2008-013-D00114), also in part, by Science Foundation Ireland grant 03/CE2/I303\_1 to Lero – the Irish Software Engineering Research Centre ([www.lero.ie](http://www.lero.ie)).

## References

- [1] P. Clements and L. M. Northrop, *Software Product Lines: Practices and Patterns*, ser. The SEI series in software engineering. Boston: Addison-Wesley, 2002.
- [2] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *ECOOP 1997*, 1997, pp. 220–242.
- [3] I. Godil and H.-A. Jacobsen, "Horizontal decomposition of prevayler," in *CASCON 2005*, 2005, pp. 83–100.
- [4] C. Kästner, S. Apel, and D. Batory, "A case study implementing features using AspectJ," in *SPLC 2007*, September 2007, pp. 223–232.
- [5] J. Liu, D. Batory, and C. Lengauer, "Feature oriented refactoring of legacy applications," in *ICSE 2006*, 2006, pp. 112–121.
- [6] K. Czarnecki and U. W. Eisenecker, *Generative Programming*. Reading, MA, USA: Addison Wesley, 2000.
- [7] K. C. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson, "Feature Oriented Domain Analysis (FODA) feasibility study," Tech. Rep., 1990.
- [8] BigLever Software, "Gears," <http://www.biglever.com/solution/product.html>. [Online]. Available: `\url{http://www.biglever.com/solution/product.html}`
- [9] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés, "Fama: Tooling a framework for the automated analysis of feature models," in *The First International Workshop on Variability Modeling of Software-Intensive Systems*, 2007.
- [10] pure-systems GmbH, "Variant management with pure::variants," 2003, <http://www.pure-systems.com>. [Online]. Available: <http://www.pure-systems.com>
- [11] G. Botterweck, K. Lee, and S. Thiel, "Automating product derivation in software product line engineering," in *Software Engineering 2009 (SE09)*, Kaiserslautern, March 2009.
- [12] K. Lee, G. Botterweck, and S. Thiel, "Aspectual separation of feature dependencies for flexible feature composition (submitted for review)," in *COMPSAC 2009*, 2009.
- [13] M. Janota and G. Botterweck, "Formal approach to integrating feature and architecture models," in *FASE 2008*, Budapest, Hungary, 29 March - 6 April 2008.
- [14] G. Botterweck, D. Nestor, A. Preussner, C. Cawley, and S. Thiel, "Towards supporting feature configuration by interactive visualisation," in *ViSPLE 2007, collocated with SPLC 2007*, Kyoto, Japan, September 10-14, 2007 2007.
- [15] K. Czarnecki, S. Helsen, and U. Eisenecker, "Formalizing cardinality-based feature models and their specialization," *Software Process Improvement and Practice*, vol. 10, no. 1, pp. 7–29, 2005.
- [16] M. L. Griss, "Implementing product-line features by composing aspects," in *SPLC 2000*, August 2000, pp. 271–288.
- [17] V. Alves, P. Matos Jr., L. Cole, P. Borba, and G. Ramalho, "Extracting and evolving mobile games product lines," in *SPLC 2005*, September 2005, pp. 70–81.
- [18] D. Batory, "Feature-oriented programming and the AHEAD tool suite," in *ICSE '04*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 702–703.
- [19] M. Mezini and K. Ostermann, "Variability management with feature-oriented programming and aspects," in *SIGSOFT 2004*, Newport Beach, CA, 2004, pp. 127–136.
- [20] N. Loughran and A. Rashid, "Framed aspects: Supporting variability and configurability for AOP," in *ICSR 2004*. Madrid, Spain: Springer, July 2004, pp. 127–140.
- [21] K. Lee, C. K. Kang, and M. Kim, "Combining feature-oriented analysis and aspect-oriented programming for product line asset development," in *SPLC 2006*, 2006.
- [22] K. Czarnecki and M. Antkiewicz, "Mapping features to models: A template approach based on superimposed variants," in *GPCE'05*, 2005.
- [23] M. Völter and I. Groher, "Product line implementation using aspect-oriented and model-driven software development," in *SPLC 2007*, Kyoto, Japan, 2007.
- [24] Eclipse Foundation, "EMP - Eclipse Modeling Project," <http://www.eclipse.org/modeling/>. [Online]. Available: <http://www.eclipse.org/modeling/>
- [25] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés, "FAMA: Tooling a framework for the automated analysis of feature models," in *VAMOS 2007*, 2007.