Representing Variability Management as a Crosscutting Concern

Rodrigo Bonifácio and Paulo Borba

June 15, 2008

Contents

Abstract

Variability management is a common challenge for Software Product Line (SPL) adoption, since developers need suitable mechanisms for describing or implementing variability that might occur at different SPL views (requirements, design, implementation, and test). In this research, we are proposing an approach for use case scenario variability management. The main goal is to improve the separation of concerns between languages used to manage variabilities and languages used to specify use case scenarios. Initial results gave evidence that, by applying our prposed approach, both representations can be understood and evolved in a separated way. We achieve such a goal by modeling variability management as a crosscutting phenomenon, for the reason that artifacts such as feature models, product configurations, and configuration knowledge crosscut each other with respect to a SPL specific member. Additionally, we argue that our proposed approach might be customized to describe variability mechanisms in other SPL artifacts, being a contribution for automatic product generation and traceability.

Chapter 1

Introduction

Software Product Line (SPL) development is a well known technique for improving systematic reuse by considering the common features of products in a shared domain [8]. From a technical point of view, SPL approach aims at customizing products from a set of reusable assets. In order to do that, developers must represent variation points in the SPL assets and describe composition rules which guide product derivation based on a specific feature selection [20].

However, *variability management* concern, due to its inherent crosscutting nature, is a common challenge in product line adoption [8, 20]. Such a crosscutting characteristic is materialized whenever a feature requires variation points to be spread in different SPL artifacts. Actually, the representation of a variant feature is often spread not only in several artifacts, but also at the different product line views (requirements, design, implementation, and test). Consequently, variability management, in the same way as the feature model and the architecture, represents a central concern in SPL development and should not be tangled with existing artifacts [20]— otherwise the SPL evolution might be comprimized. In order to mitigate this problem in source code, several authors have proposed the use of *aspect-oriented* mechanisms to better modularize the composition of common and variant assets of a product line [2, 4].

In this thesis we go beyond this composition issue. We mainly consider a more encompassing notion of variability management, including artifacts such as feature models [14, 9] and configuration knowledge [9, 20], and explain it as a crosscutting phenomenon. Our goal is to investigate the real benefitis of representing variability management as a crosscutting mechanism. More specifically, we aim to (a) present the role of each variability

management artifact (such as feature models, product configurations, and configuration knowledge); and (b) describe how they contribute to generate product specific artifacts. One key characteristic of our approach is that variability management should be separated from software engineering assets. We achieve such a goal by modeling the variability management concern as a crosscutting mechanism. Reducing the impact of changes both in variability space (introducing new features or products) and in soluction space is the main expected benefit of applying our proposed approach. As secondary benefits, we also expect to improve traceability and product generation.

Therefore, we can describe our reseach hypotheses as:

Hypotheses. A clear and formal separation between **variability management** and **software engineering assets** improves SPL evolvability, traceability, and product generation.

In a first moment, we applied this view of *variability management as crosscutting* for modularizing SPL use case scenarios. The choice of applying our approach in this context was motivated because current techniques for scenario variability management [15, 6, 11] do not present a clear separation between variability management and scenario specification. As consequence, supposing that details about product variants are tangled with use case scenarios, if one variant is removed from the product line, it would be necessary to change all related scenarios. In summary, when variability management is tangled with requirements, the evolution of a SPL is compromised, being difficult to introduce new product variants. The same is true when this kind of tangling occurs with other software assets, such as design, source code, and test artifacts.

Another problem is the lack of a formal representation for deriving product specific scenarios, which is not suitable for current SPL generative practices [17]; following the aforementioned works, it is difficult to automatically derive the requirements of a specific SPL member and to check if the specified compositions between common and variant scenarios are correct. Although the semantic composition of PLUC (Product Line Use Case) [6] is defined in [12], such an approach does not separate scenario specification from variability management, as presented in Section 2.

This report is organized as followed. First we present a motivating problem which describe, in more details, some issues regard to modularity of the current techniques for representing variabilities in SPL scenarios (Chapter 2. Then, in Chapter 3, we report the initial results of this research, presenting the contributions of each involved language (Section 3.1, the proposed vari-

ability modeling framework (Section 3.4), and the semantics of three kinds of variability in SPL scenarios (optional scenarios, ..., and parameterized scenarios). Chapter **??** presents some cases that we have evaluated and compared our approach with existing ones. Such a evaluatetion considered both Design Structure Matrices [5] and a suite of metrics that we have customized for our context [7]. Finally, we conclude by presenting a schedulling for the next activities of this research.

Chapter 2

Motivating Example

In order to customize specific products, by selecting a valid feature configuration, variantion points must be represented in the product line artifacts. Several variant notations have been proposed for use case scenarios, like Product Line Use Cases (PLUC) [6] and Product Line Use Case Modeling for Systems and Software Engineering (PLUSS) [11]. However, besides the benefits of variability support, existing works do not present a clear separation between variability management and scenario specifications. In this section we illustrate the resulting problems using the *eShop Product Line* [21] as a motivating example.

The main use cases of the *eShop Product Line* (EPL) allow the user to *Register as a Customer*, *Search for Products*, and *Buy Products*. Five variant features are described in the original specification, allowing a total of 72 applications to be derived from the product line [21]. Here, we consider extra features, such as *Update User Preferences*, which, based on the user historical data of searches and purchases, updates user preferences. Figure 2.1 presents part of the *eShop* feature model [14, 9]; the relationships between a parent and its child are categorized as: *Optional* (features that might not be selected in a specific product), *Mandatory* (features that must be selected, if the parent is also selected), *Or* (one or more subfeatures might be selected), and *Alternative* (exactly one subfeature must be selected for each product).

In the PLUSS approach, all variant steps of a scenario specification are defined in the same artifact. For example, steps 1(a) and 1(b) in Figure 2.2 are never performed together. They are alternative steps: Step 1(a) will be present only if the *Shopping Cart* feature is selected (otherwise Step

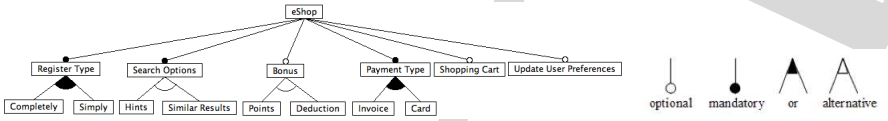

Figure 2.1: eShop feature model.

1(b) will be present)[1]. In a similar way, we have to choose between options (a) and (b) for Step 2 (depending on the *Bonus Feature* has been selected or not). Finally, Step 6 is optional and will be present only if the feature *Update User Preference* is selected. In this technique, such restrictions are documented in feature models.

Following this approach, it is hard to understand the behavior of a specific product because all possible variants are described in the same artifact. Therefore, it is not possible to reason about scenarios in a modular way. Moreover, such tangling between variant representation and scenario specification results in maintainability issues: introducing a new product variant might require changes in several points of existing artifacts. For example, including a *B2B Integration* feature, which allows the integration between partners in order to share their warehouses, might change the specification of the *Buy Product* scenario, enabling the search for product availability in remote warehouses (a new variant for Step 1) and updating a remote warehouse when the user confirms the purchase (a new variant for Step 5). Moreover, the inclusion of this new optional feature also changes the specification of the *Search for Products* scenario (the search might also be remote). In summary, since the behavior of certain features may be spread among several specifications and each specification might describe several variants, the effort needed to understand and evolve the product line might increase.

Instead of relating each variant step to a feature, PLUC introduces special tags for representing variabilities in use case scenarios. For example, the VP1 tag in Figure 2.3, which also describes the *Buy Products* scenario, denotes a variation point that might assume the values "*checkout*" or "*buy item*", depending on which product is configured. For each *alternative* or *optional* step, one tag must be defined. The actual value of each tag is specified in the *Variation Points* section of the scenario specification.

A different kind of tangling occurs in this case, since segments of the spec-

[1]Due to space constraints, we do not represent the relationships between features and PLUSS scenarios in this paper.

Id	User Action	System Response
1(a)	Select the checkout option.	Present the items in the shopping cart and the amount to be paid. The user can remove items from shopping cart.
1(b)	Select the buy product option.	Present the selected product. The user can change the quantity of item that he wants to buy. Calculate and show the amount to be paid.
2(a)	Select the confirm option.	Request bonus and payment information.
2(b)	Select the confirm option.	Request payment information.
3	Fill in the requested information and select the proceed option.	Request the shipping method and address.
4	Select the $ShipMethod$, fill in the destination address and select the proceed option.	Calculate the shipping costs.
5	Confirm the purchase.	Execute the order and send a request to the Delivery System in order to dispatch the products.
6	Select the close section option.	Register the user preferences.

Figure 2.2: Buy Products scenarios using the PLUSS notation.

ification are tangled with the variation points. Additionally, SPL members are also described using the same tag notation (see the *Product Definition* section in Figure 2.3). There is no explicit relationship between product configurations and feature models. In the example, two products (P1 and P2) are defined. The first product is implicitly configured by selecting the *Shopping Cart*, *Bonus*, and *Update User Preferences* features. The second model, on the other hand, is not configured with such features.

Since the values of alternative and optional variation points are computed based on the defined products, instead of specific features, the inclusion of a new member in the product line might require a deep review of variation points. Moreover, since the variation points and the product definitions are spread among several scenario specifications, it is hard and time

```
    Buy Products Scenario
    Main Flow
01  Select [VP1] option
02  [VP2]
03  Select the confirm option
04  [VP3]
05  Fill in the requested information and select the
    proceed option
06  Request the shipping method and address
07  Select the [VP4] shipping method, fill in the
    destination address and select the proceed option
08  Calculate the shipping costs.
09  Confirm the purchase.
10  Execute the order and sends a request to the
    Delivery System in order to dispatch the products
11  Select the close section option.
12  {[VP5] Register the user preferences.}

    Products definition:
    VP0 = (P1, P2)

    Variation points:
    VP1 = if (VP0 == P1) then (checkout)
          else (buy product)
    VP2 = if (VP0 == P1)
          then (Presents the items in the shopping
    cart...)
          else (Present the selected product.  The
    user...)
    VP3 = if (VP0 == P1)
          then ( Requests bonus and payment
    information.)
          else (Requests payment information.)
    VP4 = (Economic, Fast)
    VP5 requires (VP0 == P1)
```

Figure 2.3: Buy Products scenarios using the PLUC notation.

consuming to keep the SPL consistent. Finally, the same definitions (product configuration and variation points) often are useful to manage variabilities in other artifacts, such as design and source code. As a consequence, this approach requires the replication of such definitions in different SPL views — when the SPL evolves, changes are propagated throughout many artifacts.

In summary, both approaches are not suitable for modularizing the crosscutting nature of certain features, have poor legibility, and lead to lower maintainability. Consequently, we argue that the variability management concern should be separated from the other artifacts and used as a language for supporting product configuration and traceability.

The next section describes our approach that considers scenario vari-

ability as a composition of different artifacts. Although in this paper we focus on use case scenarios, the idea of separating product line artifacts from variability management (using feature models, product configurations, and configuration knowledge) is also applied to other SPL views.

Chapter 3

Initial Results

In order to describe our approach, we propose a framework for modeling the composition processes of scenario variabilities with feature models, product configuration, and configuration knowledge (Section 3.1). Such a framework aims to: (1) represent a clear separation between variability management and scenario specification; and (2) specify how to weave these representations in order to generate specific scenarios for a SPL member. Therefore, the main contributions of this work are

- Characterize the broader notation of variability management as a crosscutting concern and, in this way, propose an approach for representing it as an independent view of the SPL. Although this work focuses on requirements artifacts, more specifically use case scenarios, we argue that such separation is also valid for other SPL views. Actually, it has already been claimed for source code [2, 4], without considering the importance of variability artifacts.

- A framework for modeling the composition processes of scenario variability mechanisms. This framework gives a basis for describing variability mechanisms (such as scenario composition and parameterization), allowing a better understanding of each of them. In this work, such a framework is used for modeling the semantics of scenario variability mechanisms, but it might be customized for other SPL views.

- Describe three scenario variability mechanisms (selection of optional scenarios, scenario composition, scenario parameterization) using the modeling framework. Such descriptions present a more formal representation when compared to existing works; this is an important

property for supporting the automatic derivation of product specific artifacts.

Since our concept of crosscutting mechanism is based on Masuhara and Kiczales work [18], another contribution of this paper is that we apply their ideas to the languages of variability management, reinforcing the generality of their model, which was originally instantiated only for mechanisms of aspect-oriented programming languages. Based on their view of crosscutting, we can reason about variability management as a crosscutting concern, since different input specifications contribute to derive a specific SPL member.

Finally, we evaluate our approach (Section 4) by comparing it to existing works on different product lines. We also relate our work with other research topics (Section 5) and present our concluding remarks in Section 6.

3.1 Variability as Crosscutting

Aiming at representing a clear separation between variability management and scenario specification, and also to describe the weaving processes required to compose these views, we propose a modeling framework, that slightly generalizes Masuhara and Kiczales (MK) framework [18], and instantiate it for the use case and product line context. The MK framework aims to explain how different *aspect-oriented* technologies support crosscutting modularity. Each technology is modeled as a three-part description: the related weaving processes take two programs as input, which crosscut each other with respect to the resulting program or computation [18].

Similarly to the MK framework, we represent the semantics of **scenario variability management** as a weaver that takes as input four specifications (*product line use case model*, *feature model*, *product configuration*, and *configuration knowledge*) that crosscut each other with respect to the resulting product specific use case model (Figure 3.1). Combining these input languages, it is possible to represent the kinds of variability that we are interested in: *optional use cases and scenarios*, *quantified changed scenarios*, and *parameterized* scenarios.

A running example of our approach is presented in Section 3.2. After that, we describe the semantics of our weaving process. For simplicity, it was decomposed in three parts — one weaver process for each kind of variability. The semantics of those weavers (and the meta-model of the input and output languages) are described using the Haskell programming language. This leads to concise weaving process descriptions (the related

source code is available at a web site [1]) and keeps our model close to MK work, where weaving processes are specified in the Scheme programming language.

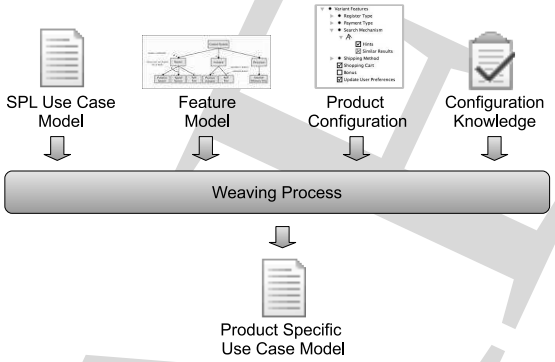

Figure 3.1: Overview of our weaving process.

3.2 Running Example

In order to explain how the input languages crosscut each other and produce a product specific use case model, here we present a running example of our approach based on the eShop Product Line (briefly introduced in Section 2). For this, several artifacts of each input language are described. Then, we present the role of each input language in respect of the weaving process.

3.2.1 SPL use case model

This artifact defines a set of scenarios that might be used to describe possible products of the SPL. Although not being directly concerned with variability management, some scenarios might be optional, might have parameters, or might change the behavior of other scenarios. Moreover, a use case scenario corresponds to a sequence of pairs *User Action* x *System Response*. A use case defines a set of scenarios; and a use case model defines a set of use cases. In this running example, we consider the following scenarios:

Buy a Product: this optional scenario (Figure 3.2) enables a customer to buy specific goods from an online shopping store. It is only available at instances of the product line that are not configured with the *Shopping Cart* and *Bonus* features. This scenario starts from the IDLE special state

(does not extend the behavior of an existing scenario) and finishes at the Step P1 of the *Proceed to Purchase* scenario (which is described later). The clauses *From step* and *To step* are used for describing the possible starting and ending points of execution.

```
Description:  Buy a specific product
From step:  IDLE
To step:  P1
```

Id	User Action	System Response
B1	Select the buy product option.	Present the selected product. The user can change the quantity of items he wants to buy. Calculate and show the amount to be paid.
B2	Select the confirm option.	Request payment information.

Figure 3.2: Buy product scenario.

Buy Products with Shopping Cart and Bonus: this optional scenario (Figure 3.3) allows the purchasing of products that have been previously added to a customer shopping cart. It changes the behavior of the *Buy a Product* scenario by replacing its first two steps and by introducing the specific behavior required by the *Shopping Cart* and *Bonus* features. This scenario also starts from the IDLE state (*From step* clause) and finishes at Step P1 of *Proceed to Purchase* (*To step* clause). This behavior is required for products that are configured with *Shopping Cart* and *Bonus* features.

```
Description:  Buy products using a shopping-cart
From step:  IDLE
To step:  P1
```

Id	User Action	System Response
V1	Select the checkout option.	Present the items in the shopping cart and the amount to be paid. The user can remove items from the shopping cart.
V2	Select the confirm option.	Request bonus and payment information.

Figure 3.3: Buy products with shopping cart scenario.

Proceed to Purchase: this mandatory scenario (Figure 3.4) specifies the common behavior that is required for confirming a purchase. Instances of the product line must be configured with this scenario. Although it can be started either after Step B2 (from *Buy a Product* scenario) or after Step V2 (from *Buy Products with Shopping Cart* scenario), just one of the paths can be available at a specific product. It is important to notice that the *From step* and *To step* clauses are used for composing, in a quantified way, different scenario configurations without specifying all possible variants in a single artifact (as suggested in the PLUSS notation). Moreover, notice that a parameter *ShipMethod* is referenced in Step P2 of Figure 3.4. The use of this parameter (notation also supported in PLUSS and PLUC) allows the reuse of this specification for different kinds of *ship method* configurations.

Description: Proceed to purchase
From step: B2, V2
To step: END

Id	User Action	System Response
P1	Fill in the requested information and select the proceed option.	Request the shipping method and address.
P2	Select one of the available ship methods (<ShipMethod>), fill in the destination address and proceed.	Calculate the shipping costs.
P3	Confirm the purchase.	Execute the order and send a request to the Delivery System to dispatch the products. [RegisterPreference]

Figure 3.4: Proceed to purchase scenario.

Search for Products: this mandatory scenario allows the user to search for products. In order to save space, we are only presenting Step S3, which performs a search based on the input criteria (Figure 3.5). This step is annotated with the mark [**RegisterPreference**], exposing it as a possible extension point for the behavior of *Register User Preferences*. The same annotation was used in the Step P3 of *Proceed to Purchase* (Figure 3.4). Such annotations can be referenced in the *from step* and *to step* clauses, reducing problems that might occur by changing step ids.

Register User Preferences: this optional scenario updates the user preferences based on the buy and search products use cases. Its behavior can be started at each step that is marked with the [**RegisterPreference**]

Description: Search for Products scenario
From step: IDLE
To step: END

Id	User Action	System Response
...	...	...
S3	Inform the search criteria.	Retrieve the products that satisfy the search criteria. Show a list with the resulting products. [RegisterPreference]

Figure 3.5: Search for products scenario.

(see the *from step* clause) annotation and is available in products that are configured with the *Update User Preferences* feature.

Description: Register user preferences.
From step: [RegisterPreference]
To step: END

Id	User Action	System Response
R1	-	Update the preferences based on the search results or purchased items.

Figure 3.6: Register user preferences.

In this running example, we described several scenarios as being optional or mandatory. It is important to observe that, in our approach, this kind of information is not specified in scenario documents. Actually, it is necessary to consider the relationships between scenarios and features in order to realize which configurations require a specific scenario. As we said at the beginning of this section, each artifact (feature model, product configuration, configuration knowledge, and use case model) provides a specific contribution to the definition of a SPL member.

3.2.2 Feature model

We have introduced part of this artifact for the eShop product line in Section 2. However, here we present only the features required (Figure 3.7) for understanding the running example.

Based on the feature model of Figure 3.7, the *Shopping Cart*, *Bonus* and

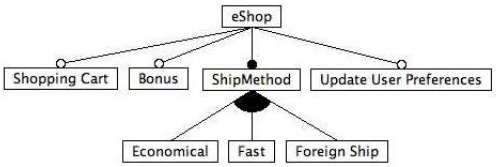

Figure 3.7: Subset of eShop feature model.

Update User Preferences features are optional; on the other hand, the *Ship Method* feature is mandatory and all products have to be configured with at least one of its child. A short review about the feature model notation was presented in Section 2. More details can be found elsewhere [14, 9].

3.2.3 Product configuration

This artifact is used for identifying which features were selected in order to compose a specific member of a product line. Each product configuration should conform to a feature model (the selected features should obey the feature model relationships and constraints). Two possible configurations are presented in Figure 3.8. We represent these configurations as a tree, highlighting which features were selected. Such a representation was created using the *Feature Modeling Plugin* [3]

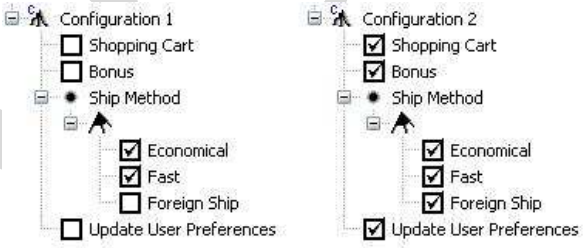

Figure 3.8: Examples of product configurations.

The first configuration (on the left side of the Figure 3.8) defines a product that has no support for shopping cart, bonus and preferences update. Additionally, it supports only the economical and fast ship methods. The second configuration selects all possible features. In order to select the required assets (scenarios, classes and aspects, test cases) for a specific product

configuration, it is necessary to relate features to them. This is the role of the configuration knowledge as an input artifact of our weaving process.

3.2.4 Configuration knowledge

This artifact is used for relating feature expressions to assets that must be assembled in a given product. Such artifacts allow, during product engineering, the automatic selection of assets that are required for a specific product configuration.

Table 3.1 presents a configuration knowledge for the running example, enforcing that *Proceed to Purchase* and *Search for Products* are mandatory scenarios, since they are related to the root feature of eShop product line; *Buy a Product* scenario is used in the composition of products that have not been configured with both *Shopping Cart* and *Bonus* features — if both features were selected for a product, it will be configured with *Buy Product with Cart* scenario; and finally, Table 3.1 states that *Register User Preference* scenario is not used in composition unless the *Update Preference* feature is selected.

Table 3.1: eShop configuration knowledge

Expression	**Required Artifacts**
eShop	Proceed to Purchase Search for Products ...
not (Cart **and** Bonus)	Buy a Product
Cart **and** Bonus	Buy Products with Cart
Update Preferences	Register user Preferences
...	...

In what follows, we describe a high level view of the weaving process that combines the input languages in order to manage scenario variability. Then, in Section 3.4 we present its semantics in terms of our modeling framework.

3.3 Weaving process

The weaving process represented in Figure 3.1 is responsible for performing the following activities:

Validation activity: This activity is responsible for checking if a product configuration is a valid instance of the feature model. If the product configuration is valid (it conforms to the relationship cardinalities and constraints of the feature model), the process might proceed.

Product derivation activity: This activity takes as input a (valid) product configuration and a configuration knowledge. Each feature expression of the configuration knowledge is checked against the product configuration. If the expression is satisfied, the related scenarios are assembled as the result of this activity. For the running example, Table 3.2 shows the assembled scenarios for the configurations depicted in Figure 3.8.

Table 3.2: Assembled scenarios

Configuration	**Assembled scenarios**
Configuration 1	Proceed to Purchase
	Search for Products
	Buy a Product
	...
Configuration 2	Proceed to Purchase
	Search for Products
	Buy Products with Cart
	Register User Preferences
	...

Scenario composition activity: This activity is responsible for composing the scenarios assembled for a specific product configuration. Therefore, the resulting scenarios of the previous activity, which crosscut each other based on the *From step* and *To step clauses*, are woven. The result is a use case model with complete paths (all *From step* and *To step* clauses are resolved).

The complete path is a high level representation, which uses the same constructions of the use case model (scenarios), and is illustrated here as a graph, where each node is labeled with a step id. For example, Figure 3.9 depicts the complete paths for the first and second configurations of our running example. In the left side of the figure, the composition of *Buy a Product* with *Proceed to Purchase* (branch labeled as B1, B2, P1, P2, P3) and *Search for Product* (branch labeled as S1, S2, S3) scenarios are presented. Contrasting, on the right side of the figure, the results of this activity are presented for the second configuration. In this case, steps B1

and B2 have been replaced by steps V1 and V2 (because *Shopping Cart* and *Bonus* features are selected) and the step R1 is introduced after steps P3 and S3 (because *Update User Preferences* is selected in this configuration).

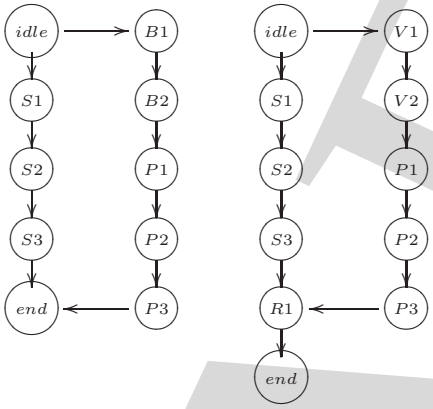

Figure 3.9: Complete paths represented as a graph

Binding parameters activity: This activity weaves scenarios and product configurations in order to resolve all scenario parameters. For example, step P2 in Figure 3.4 has a reference to the *ShipMethod* parameter, whose domain values are defined in the product configuration. For instance, in the first configuration depicted in Figure 3.8, the parameter *ShipMethod* might assume the values *Economical* or *Fast*. In order to reduce the coupling between scenario specifications and feature model, a mapping is used for relating scenario parameters to features. In fact, this mapping is another input artifact of our modeling framework; but it was not represented in Figure 3.1 because it was just introduced for avoiding explicit dependencies between feature and use case models. Next, we introduce the modeling framework used to formally describe the weaving processes just presented.

3.4 Modeling Framework

As mentioned before, the semantics of crosscutting, used for representing our variability management weaving process, is based on the Masuhara and Kiczales work [18]. It is important to notice that their requirement for characterizing a mechanism as crosscutting is fulfilled by our approach, since different specifications contribute to the definition of a specific SPL member. As a consequence, due to its crosscutting nature, the modeling frame-

work proposed in [18] is suitable for representing variability management compositions.

For simplicity, our weaving process is described as being composed by three major weavers: *product derivation weaver*, *scenario composition weaver*, and *bind parameters weaver*. As a customization of MK work, our modeling framework represents each weaver as an 6-tuple (Eq. 3.1 and Table 3.3), highlighting the contribution of each input language in the composition processes.

$$W = \{o, o_{jp}, L, L_{id}, L_{eff}, L_{mod}\}, \tag{3.1}$$

Table 3.3: Modeling framework elements.

Element	**Description**
o	Output language used for describing the results of the weaving process
o_{jp}	Set of join points in the output language
L	Set of languages used for describing the input specifications
$L_{ID}(l)$	Set of constructions in each input language l, used for identifying the output join points
$L_{EFF}(l)$	For each input language l, this element represent the effect of its constructions in the weaving process
$L_{MOD}(l)$	Set of modular unities of each input language l

We represent each weaver by filling in the six parameters of our 6-tuple representation, by providing an reference implementation for each weaver, and by stating how elements of the weaver implementation correspond to elements of the model.

3.5 Product derivation weaver

This weaver is responsible for selecting artifacts based on specific product configurations. As a consequence, it implements the first two activities of

our variability management approach: validating a product configuration against a feature model and selecting a subset of the SPL assets. Although in this paper we are focusing only in the selection of scenarios that should be assembled in specific instances of the SPL, this weaver can be easily extended for managing variabilities in other kinds of assets (aiming at selecting design elements, source components, and test cases).

For instance, applying this weaver for combining the eShop use case model, feature model , and configuration knowledge with the configuration depicted in right side of Figure 3.8 will result in the selection of *Buy Products with Cart*, *Proceed to Purchase*, *Search for Products*, and *Register User Preferences* scenarios. This weaver is implemented by the function *pdWeaver* (Listing 3.1) and takes as input a *SPL use case model* (UCM), a *feature model* (FM), a *product configuration* (PC), and a *configuration knowledge* (CK).

Initially, this function verifies if the product configuration is a well formed instance of the feature model (*validInstance* function) — if it is not the case, an *InvalidProduct* error is thrown. Then, the IDs of selected scenarios are computed by the *configure* function. This is done by evaluating which feature expressions, defined in the list elements (x:xs) of configuration knowledge, are valid for the specific product instance (*eval* function). Finally, given the resulting list of scenario IDs, the function *retrieveArtifacts* returns the product specific scenarios. As a consequence, we can realize two levels of crosscutting in this weaver. First, the feature model, the product configuration, and the configuration knowledge crosscut each other in order to contribute to the list of valid scenario IDs composition. Then, the resulting list of scenario IDs crosscuts with the use case model for selecting the product specific scenarios.

Listing 3.1: Product derivation weaver function

```
pdWeaver :: UCM -> FM -> PC -> CK -> ScenarioList
pdWeaver ucm fm pc ck =
 if not (validInstance fm pc)
 then error InvalidProduct
 else retrieveScenarios ucm (configure pc ck)

configure :: PC -> CK -> ListOfScenarioId
configure pc (CK []) = []
configure pc (CK (x:xs)) =
 if (eval pc (expression x))
  then (artifacts x) ++ (configure pc (CK xs))
```

```
else configure pc (CK xs)
```

The model of the *Product Derivation Waver*, in terms of the framework, is shown in Table 3.4. The *pdWeaver* function is used to argue that the model is realizable and appropriate [18]. We achieve this by matching the model elements to corresponding parameters and auxiliary functions in the implementation code. Therefore, the input languages UCM, FM, CK, and PC are represented as different parameters of the *pdWeaver* function. An instance of the UCM corresponds to the specification of all SPL scenarios. A FM instance is only responsible for declaring the SPL features and the relationships between them; as a consequence, there is no coupling between FMs and UCMs. Instead, relationships between features and artifacts are documented in the configuration knowledge. Finally, the PC specifies which features were selected for a specific product.

Table 3.4: Model of Product Derivation

Element	**Description**
o	Product specific scenarios (list of scenarios)
o_{jp}	Scenario declarations
L	{UCM, FM, CK, PC}
UCM_{ID}	SPL scenarios
FM_{ID}	SPL features
CK_{ID}	Feature expressions and scenario IDs
PC_{ID}	Product specific feature selection
UCM_{EFF}	Provides declaration of scenarios
FM_{EFF}	Checks if a SPL instance is well formed
CK_{EFF}	Identifies selected artifacts
PC_{EFF}	Triggers scenario selection
UCM_{MOD}	Scenario
FM_{MOD}	Feature
CK_{MOD}	Each pair $(feature\ expression, artifact\ list)$
PC_{MOD}	Feature

The UCM has a greater importance over the other input languages

(UCM_{EFF}), since it declares the parts that compose the product specific scenarios (the output of this weaver process generated by the *pdWeaver* function). These scenarios (UCM_{ID}) are used in the *retrieveScenarios* function in order to identify which artifacts will be assembled in the final product.

In order to identify which artifacts are required for a specific product, the *configure* function (CK_{EFF}) checks the feature expression (CK_{ID}) against the product specific features (PC_{ID}). The effect of FM in this weaver (FM_{EFF}) is to check if the PC is well formed. Such evaluation is implemented by the *validInstance* function and considers the PC feature selection (PC_{EFF}).

3.6 Scenario composition weaver

This weaver is responsible for the third activity of our variability management approach. It aims at composing variant scenarios of a use case and is applied whenever a use case scenario supports different execution paths. This mechanism takes as input the product specific use case model (a list of scenarios). Each scenario, often partially specified, is then composed in order to generate concrete specifications.

As discussed in Section 3.2.1, a variant scenario might refer to steps either in basic or other variant scenarios. In order to compute the complete paths defined by a scenario, we need to compose the events that precede all steps referenced by its *From step clause* (up to the IDLE step), followed by its own steps, and then by all events that follow all of the steps referenced by its *To step clause* (down to the END step).

For instance, consider a product configured with the features *Shopping Cart* and *Bonus*, which requires the *Buy Products with Cart* scenario, and with the feature *Update User Preferences*. Referring to Figure 3.3, the *Buy Product with Cart* scenario starts from the IDLE state (*From step* clause) and then, after its own flow of events, goes to Step P1 of *Proceed to Purchase* (see the *To step* clause). In a similar way, Figure 3.6 depicts that *Register User Preferences* scenario starts from any step that is marked with the *RegisterPreferences* annotation (for example, Step P3 of of *Proceed to Purchase*). In this context, the result of applying the composition scenario weaver is a concrete path of execution for this configuration, that can be represented as this sequence of step ids:

$$< IDLE, V1, V2, P1, P2.ShipMethod, P3, R1, END >.$$

Note that this sequence still has the *ShipMethod* parameter, referred in

Listing 3.2: Scenario composition weaver function

```
scWeaver :: ScenarioList -> [StepList]
scWeaver scenarioList = [completePaths scenarioList s | s <- scenarioList]

completePaths :: ScenarioList -> Scenario -> [StepList]
completePaths ucm scn =
 (fromList ucm (match ucm (fromSteps scn)) +++ [stepsOf scn]) +++
(toList ucm (match ucm (toSteps scn)))
```

Step P2 of *Proceed to Purchase* scenario. The *Binding parameter weaver*, discussed in the next section, is responsible for resolving parameters in the final product specification.

The *Scenario Composition Weaver* is implemented by the *scWeaver* function (Listing 3.2), which takes as input the product specific use case model (a list of scenarios computed by the previous weaver). The *scWeaver* function computes the complete paths of each scenario by calling, recursively, the *completePaths* function. This function (lines 5-6 in Listing 3.2) takes as input the product specific use case model (*ucm*) and a specific scenario (*scn*); and returns all complete paths (a list of *step lists*) of *scn*. The function *fromList* (called at line 7) is used to compose all complete paths extracted from the *From step clause*. In a similar way, the function *toList* (called at line 7) is used to compose all complete paths extracted from the *To step clause*. The *match* function (also called at line 7), retrieves all the steps in *ucm* that satisfy all *step references* in *From step* or *To step* clauses.

The model of this weaver is in Table 3.5. The output (*o* element of our modeling framework) is the complete paths of the product specific scenarios, computed directly from *scWeaver* function. Therefore, the input language (L) corresponds to the product specific scenarios, related to the *scWeaver* parameters. The join points are modeled as the final scenarios and steps in the output language. They result from the composition of partial scenarios by means of *From steps* and *To steps* clauses (L_{ID}). The effect of the input languages (L_{EFF}) in the composition process is to combine product specific scenarios that, before this activity, did not define a concrete flow of events. As a consequence, the *match* function plays a fundamental role in this process, retrieving the steps in the use case model that satisfies the *From step* and *To step* clauses.

Table 3.5: Model of Scenario Composition Weaver

Element	Description
o	List of composed scenarios
o_{jp}	Scenarios and steps of scenarios
L	{Product specific scenarios (list of scenarios)}
L_{ID}	From step and to step clauses
L_{EFF}	Defines abstract scenarios
L_{MOD}	Scenarios

3.7 Bind parameters weaver

This weaver is responsible for the third activity of our variability management process. Parameters are used in scenario specifications in order to create reusable requirements. For instance, Figure 3.4 depicts the *Proceed to Purchase* scenario that can be reused for different *ship methods*. Without this parameterized specification, and aiming, for example, at automatically generating a test case suite with a good coverage, it would be necessary to create a specification for each kind of shipment method.

This weaver takes into consideration *scenario specifications* and *product configurations*, which defines the domain values of a parameter. Thus, in order to reduce the coupling between scenarios and features, we propose a mapping that relate them. A constraint must be obeyed in this mapping: features related to parameters must be either an **alternative feature** or an **or feature** [14, 10, 9].

The implementation of this weaver consists of calls to the *bpWeaver* function (Listing 3.3) for each step available in the product specific scenarios or complete paths. This function (lines 1-5 of Listing 3.3) takes as input a mapping (m), which relates a scenario parameter to a feature; a product configuration (pc), which defines the domain values of parameters (expressed as the feature selection); and a step (s) that may be parameterized. Then, it replaces all parameters from s, returning it as a suitable representation with the corresponding parameter values. Each text between the symbols "<" and ">" (defined in the user action or system response of a step) is treated as a parameter and must be defined in the mapping.

Listing 3.3: Bind parameter weaver function

```
bpWeaver :: Mapping -> PC -> Step -> Step
bpWeaver m pc s =
 if (length (extractParameters (s)) == 0)
  then s
  else replaceParameterValues m pc s
```

For example, if a product is configured with either *Economical* and *Fast* shipment methods, the result of applying this weaver for the Step P2 of the *Proceed to Purchase* scenario will result in the representation (*Economical or Fast*) in each place that the parameter *ShipMethod* is referred. Table 3.6 describes the Bind Parameters model. This weaver just resolves parameters in scenario specifications. Therefore, its output language is also a list of scenarios; but with resolved parameters (the join points).

Table 3.6: Model of Bind Parameters Weaver

Element	**Description**
o	List of scenarios with resolved parameters
o_{jp}	Each resolved parameter
L	{UCM, PC, Mapping}
UCM_{ID}	Parameterized steps
PC_{ID}	Selected features related to parameters
$Mapping_{ID}$	Key entries (parameter name) of the mapping
UCM_{EFF}	Declares parameterized scenarios
PC_{EFF}	Defines the domain value of parameters
$Mapping_{EFF}$	Relates parameters to features
UCM_{MOD}	Use case scenarios
PC_{MOD}	Selected features
$Mapping_{EFF}$	Each entry in the mapping

The use case model (UCM) defines the list of scenarios that might be parameterized (UCM_{EFF}). Each step of a scenario (UCM_{ID}), indeed, contributes to the definition of one join point in this weaver. The other contributions come from the configuration knowledge (CK), in the sense that the domain values of a parameter is defined (CK_{EFF}) in the product specific

features; and from the mapping (m parameter of the *bind* function) that is used for relating parameters to features. In what follows, we present an evaluation of our approach based on the specification of SPLs in different domains.

Chapter 4

Evaluation

We have applied our approach to three SPLs: the eShop Product Line, introduced in Section 2; the **Pedagogical Product Line (PPL)** [19], which was proposed for learning about and experimenting with software product lines and focus on the arcade game domain; and the **Multimedia Message Product Line (MMS-PL)**, which allows the assembling of specific products for creating, sending, and receiving multimedia messages (MMS) in mobile phones.

Based on the last application of our approach, we noticed the benefits of a clear separation between variability management and scenario specification [7], compared our approach with the PLUC and PLUSS techniques. This comparison uses Design Structure Matrices (DSMs), a suite of metrics for quantifying modularity and complexity of specifications, and observations of the effort required to introduce SPL increments (such as new variants or products). It is important to note that we did not formalize variability management as crosscutting in our previous work [7]; we just report on the benefits related to the *separation of concerns* (SoC) claimed in this paper.

4.1 DSM Analysis

For now, let us reproduce the DSMs to understand the benefits of SoC in variability management. DSMs is an interesting and simple tool for visualizing dependencies between design decisions [5]. Such decisions appear in the rows and columns of a matrix. We can identify a dependency by observing the columns in a given row [5]. For example, the first row in Figure 4.1 indicates that the task of creating the feature model depends on

the task of creating the use case model. The first can not be independently performed after the second. As presented in Section 2, the PLUC approach describes product instances and variability space at specific sections of use cases. Therefore, it is not possible to evolve variability management (introducing new features, products, or relations between features and artifacts) without reviewing the use case model. This is expressed in the non modular DSM of Figure 4.1, which depicts cyclical dependencies between use cases and variability management artifacts.

		1	2	3	4
1	Feature model		x		
2	Use case model	x		x	x
3	Product configurations	x	x		
4	Configuration knowledge	x	x		

Figure 4.1: DSM Analysis of PLUC

Our approach reduces the dependencies between variability management and scenario specifications (Figure 4.2). For instance, changes in feature model or new definitions of products do not require changes in the use case model. This clear separation is also necessary in source code, as claimed in [2, 4], and might be required in other artifacts too. Notice that the proposed mapping artifact, used to relate use cases parameters and features (Section 3.7), can be considered as a design rule, since it primarily aims at decoupling use cases and features.

		1	2	3	4	5
1	Feature model					
2	Mapping	x				
3	Use case model		x			
4	SPL instances	x				
5	Configuration model	x		x		

Figure 4.2: DSM Analysis of the proposed approach

4.2 Quantitative Analysis

For confirming the observations derived from the DSM analysis, we applied the metric suite for the *MMS* and *Pedagogical* product lines. We compared our specification of *MMS* product line to the specifications we wrote for both PLUC and PLUSS techniques [1]. In a similar way, we compared our

specification of the *Pedagogical* product line to a specification that was sent to us by the authors of the PLUSS approach.

The metric suite, used in what follows, was adapted from [13] for both product line and use cases contexts. It quantifies feature modularity and use case complexity. The proposed modularity metrics quantify three types of relations involving features and use cases. First, *Feature Diffusion over Use Cases* (FDU) is used for quantifying how many use cases are affected by a specific feature. On the other hand, *Number of Features per Use Case* (NFU) is used for quantifying how many features are tangled within a specific use case. We assume that each use case should focus in its primary goal, although several features might be related to the primary goal of a use case. Finally, we applied the metric *Feature Diffusion over Scenarios* (FDS) in order to quantify how many internal use case members (scenarios) are necessary for the materialization of a specific feature.

Moreover, we used three metrics related to complexity. The first one, *Vocabulary Size*, quantifies the number of use cases (VSU) and scenarios (VSS) required by each of the evaluated approaches. The second one, *Steps of Specification* (SS), is related to the size of each scenario and identifies how many pairs *User action* x *System response* compose a specific scenario. Additionally, we also relate modularity to complexity by applying *Features and Steps of Specification* (FSS), which counts the number of steps of specification whose main purpose is to describe the behavior of a feature. A complete description of these metrics can be found elsewhere [7]. Next we present the quantitative analysis of the *MMS* and *Pedagogical* product lines using these metrics.

4.2.1 MMS Product Line

As explained earlier, the MMS product line, based on a real case study from Motorola phones, enables the customization of multimedia message applications. The primary goal of each one of these applications is to create and send messages with embedded multimedia content (image, audio, video) [7]. We have specified the MMS product line using three techniques: our notation, PLUC, and PLUSS approaches. After that, we evaluated these specifications observing the metric suite just presented. The results of this evaluation are summarized in Table 4.1.

Since PLUC and PLUSS do not allow a scenario to crosscut other scenarios in different use cases, it is difficult to modularize features. The result is that, when comparing to the crosscutting approach, features, on the average, are more diffused (FDU, FDS, and FSS metrics) and use cases are less

Table 4.1: MMS quantitative evaluation

	PLUC	PLUSS	Crosscutting
Mean value of FDU	3.5	3.5	2
Mean value of FDS	6.25	5	4.25
Mean value of NFU	2	2	1
Mean value of FSS	12	11	10.25
VSU	5	5	7
VSS	27	24	23
SS	75	64	56

concise (NFU) in these approaches (Table 4.1). The crosscutting approach, in contrast, allows the composition of scenarios through *from steps* and *to steps* clauses.

The lack of crosscutting mechanisms for composing scenarios of different use cases also results in greater complexity, when observing the *Steps of Specification* metric. Although our approach resulted in a greater number of use cases, we improve the specification reuse, since scenarios that crosscut different use cases can be specified without duplication.

4.2.2 Pedagogical Product Line

We compared our approach against two specifications of the Pedagogical Product Line (PPL): the original one [19], proposed by the Software Engineering Institute, and a specification that was sent to us by the authors of the PLUSS technique.

The original specification of PPL is already well modularized, since its features, in general, are not crosscutting among different use cases (see modularity metrics in Table 4.2). Moreover, another characteristic of PPL is that several features are related to qualities that do not cause effect into use case specifications.

Table 4.2: PPL quantitative evaluation

	SEI	PLUSS	Crosscutting
Mean value of FDU	1.83	1.3	1.2
Mean value of FDS	3.3	3	2.5
Mean value of NFU	2	2	1.4
Mean value of FSS	3.8	3.5	3
VSU	12	7	6
VSS	25	19	16
SS	44	38	32

Still in this context, our approach also achieves some improvements in the resulting specifications. The main factor for these improvements in the *Pedagogical* product line was the error handling modularization. By applying our approach, all behavior related to the *error handling* concern is specified in a single use case. The composition of *error handling* with the basic scenarios was done by means of annotations attributed to the corresponding steps.

Therefore, both in the SEI and PLUSS specifications of the *Pedagogical* product line, several use cases were specified with scenarios for handling this kind of exception. As a consequence, we achieve a reduction (almost 20%) in the number of specification steps (SS in Table 4.2) when comparing to the PLUSS approach. It is important to notice that this reduction of size does not compromise the requirement coverage; but actually it represents an improvement in the specification reuse.

For concluding our Pedagogical product line analysis, we can realize, based on Table 4.2, that our approach achieved real benefits only in the complexity metrics. This result comes from the non crosscutting nature of PPL features. Comparing to the MMS product line results, we argue that the benefits of applying our approach should be greater in contexts where features can not be well modularized using existing techniques.

Chapter 5

Related Work

Masuhara and Kiczales (MK) proposed a modeling framework for characterizing implementation techniques as *crosscutting mechanisms* [18]. A critical property of their model is that two different input programs crosscut each other with respect to the resulting program or computation. In this paper, we applied the MK framework for representing *variability management* as a crosscutting mechanism. Improving modularity is the main reason for representing variability management as a crosscutting concern. Actually, we improved both changeability and modular reason by applying our approach for managing variabilities in use case scenarios (Section 4).

Pohl et al. argue that variability management should not be integrated into existing models [20]. Their proposed Orthogonal Variability Model (OVM) describes traceability links between variation points and the conceptual models of a SPL. Our approach also decouple variability concern. However, we describe, in more details, its semantics as a crosscutting mechanism. Additionally, we make clear how to relate SPL features to software engineering assets by means of the configuration model. In our approach, such a model can be used for reasoning about traceability.

Finally, several approaches have been proposed for representing scenario variability [16, 15, 11, 6]. However, in this paper we only compared our crosscutting approach with PLUC and PLUSS techniques because they encompass a broad range of SoC between variability management and scenario specification. PLUC presents the lowest level of modularity, since almost all information related to variability is tangled within use cases. Although PLUSS partially reduces such coupling, by considering the importance of feature modeling, some dependences from feature to use cases are still present. These dependences are avoided in our approach.

Chapter 6

Conclusions

In this paper we formally described variability management as a crosscutting mechanism, considering the contribution of different input languages that crosscut each other for deriving specific members of a product line. We applied this notion of variability management in the context of use case scenarios, achieving several benefits. First, applying our approach resulted in a clear separation of concerns between variability and scenario specifications, allowing both representations to evolve independently. Second, we achieved a reduction in the size of specifications by composing scenarios in a quantified way. Finally, the formal specification allowed us to perform several automatic verification in the composition process. This is an interesting characteristic of our approach, since it can be used for finding inconsistencies in the final products. Although our modeling framework was instantiated for representing scenario variabilities, we believe that it could be applied in other SPL artifacts. Particularly, optional and parameterized artifacts are also relevant for non-functional requirements and test cases. Additionally, our notion of crosscutting was based on a work that states *what* defines a source code technology as supporting crosscutting modularity. Therefore, we argue that representing variabilities in source code, using our modeling framework, is straightforward.

Bibliography

[1] Software productivity group, online: http://www.cin.ufpe.br/spg (2007).

[2] V. Alves, A. C. Neto, S. Soares, G. Santos, F. Calheiros, V. Nepomuceno, D. Pires, J. Leal, and P. Borba. From conditional compilation to aspects: A case study in software product lines migration. In *1st Workshop on Aspect-Oriented Product Line Engineering (AOPLE)*, Portland, USA, Oct 2006.

[3] M. Antkiewicz and K. Czarnecki. Featureplugin: feature modeling plug-in for eclipse. In *eclipse '04: Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*, pages 67–72, New York, NY, USA, 2004. ACM.

[4] S. Apel, T. Leich, and G. Saake. Aspectual mixin layers: aspects and features in concert. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 122–131, New York, NY, USA, 2006. ACM.

[5] C. Baldwin and K. Clark. *Design Rules The Power of Modularity*, volume 1. The MIT Press, first edition edition, 2000.

[6] A. Bertolino and S. Gnesi. Use case-based testing of product lines. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 355–358, New York, NY, USA, 2003. ACM.

[7] R. Bonifácio, P. Borba, and S. Soares. On the benefits of variability management as crosscutting. In *Early Aspects*, Brussels, Belgium, mar 2008.

[8] P. Clements and L. Northrop. *Software product lines: practices and patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

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

[10] K. Czarnecki and C. Kim. Cardinality-based feature modeling and constraints: A progress report. *International Workshop on Software Factories*, 2005.

[11] M. Eriksson, J. Borstler, and K. Borg. The pluss approach - domain modeling with features, use cases and use case realizations. In *International Conference on Software Product Lines*, pages 33–44. LNCS, 2005.

[12] A. Fantechi, S. Gnesi, G. Lami, and E. Nesti. A Methodology for the Derivation and Verification of Use Cases for Product Lines. *Proceedings of the International Conference on Software Product Lines, Lecture Notes in Computer Science*, 3154:255–265, 2004.

[13] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, and A. von Staa. Modularizing design patterns with aspects: a quantitative study. *Aspect-oriented software development: Proceedings of the 4 th international conference on Aspect-oriented software development*, 14(18):3–14, 2005.

[14] R. Gheyi, T. Massoni, and P. Borba. A theory for feature models in alloy. In *First Alloy Workshop*, pages 71–80, Portland, United States, nov 2006.

[15] M. L. Griss, J. Favaro, and M. d' Alessandro. Integrating feature modeling with the rseb. In *ICSR'98*, page 76, Washington, DC, USA, 1998. IEEE Computer Society.

[16] I. Jacobson, M. Griss, and P. Jonsson. *Software reuse: architecture, process and organization for business success*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1997.

[17] C. W. Krueger. New methods in software product line practice. *Commun. ACM*, 49(12):37–40, 2006.

[18] H. Masuhara and G. Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In *European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science, pages 2–28. Springer, 2003.

[19] J. McGregor. Pedagogical product line. http://www.sei.cmu.edu/productlines/ppl/index.html (2008).

[20] K. Phol and G. B. ands Frank Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[21] K. Pohl and A. Metzger. The eshop product line. online: http://www.sei.cmu.edu/splc2006/eShop.pdf.