Scenario Variability as Crosscutting

Rodrigo Bonifácio and Paulo Borba

Informatics Center
Federal University of Pernambuco
Recife, Brazil
{rba2,phmb}@cin.ufpe.br,
WWW home page: http://www.cin.ufpe.br

Abstract. Variability management is one common challenge for Software Product Line (SPL) adoption. Specifically, the community needs suitable mechanisms for describing or implementing each kind of variability that might occurs at the different SPL views (requirement, design, implementation, and test). In this paper, we present a framework for modeling scenario variability mechanisms, enabling better separation of concerns between languages used to manage variabilities and languages used to specify use case scenarios. The result is that both representations can be understood and evolved in a separated way. We achieve such goal by modeling variability management as a crosscutting phenomenon, since features often affect different points of each SPL view. Additionally, we believe that our proposed framework might be customized to describe variability mechanisms in other SPL artifacts, being a contribution for automatic product generation and traceability.

1 Introduction

The support for variation points enables product customization from a set of reusable assets [27]. However, variability management, due to its inherent crosscutting nature, is a common challenge in software product line (SPL) adoption [7, 27]. Such crosscutting characteristic is materialized whenever a feature requires variation points to be spread in different SPL artifacts. Actually, the representation of a variant feature often is spread not only in several artifacts, but also at the different product line views (requirement, design, implementation, and test). In this case, the variability management, in the same way as the feature model and the architecture, represents a central concern in the SPL development and should not be tangled with existing artifacts [27].

However, supporting techniques for scenario variability management [17, 4, 11] do not present a clear separation related to the role of each involved language. The result is that variability mechanisms become tangled with scenario specifications. An example of this problem occurs when a scenario specification describes all of its possible variants. If one variant is removed from the product line, it would be necessary to change all related scenarios. In summary, it is difficult to evolve both of these representations. Another problem is the lack

of a formal representation of the composition processes used to derive product specific scenarios, which is not suitable for current SPL generative practices. Following the aforementioned works, it is difficult to automatically derive the requirements of a specific SPL member. Although the semantic composition of PLUC (Product Line Use Case) [4] had been defined in [12], such approach does not separate scenario specification from variability management. In order to explain in more details the problems addressed by this work, a motivating example is presented in Section 2.

In this paper, we describe a framework for modeling the composition process of scenario variabilities (Section 3). Such 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. The main contributions of this work are:

- Characterize variability management as a crosscutting concern and, in this way, enforce that it must be represented as an independent view of the SPL. Although this work focus on requirement artifacts, more specifically use case scenarios, we argue that such separation is also valid for other SPL views.
- Present a framework for modeling the composition process of scenario variability mechanisms. This framework is proposed as a notation for describing variability mechanisms, which allows a better understanding of each technique. In this work, such framework is used for modeling the semantics of scenario variation mechanisms, but it might be customized for other SPL views.
- Describe each scenario variation mechanism using the modeling framework. Such descriptions present a more formal representation when compared to existing works, an important property since there is a need to automatically derive product specific artifacts.

We evaluate our approach (Section 4) by comparing it to existing works and observing a set of three criteria: support for the different kinds of variability, separation of concerns, and design expressiveness. We also relate our work with other research topics (Section 5) and present our concluding remarks in Section 6.

2 Motivating Example

In order to customize specific products, by selecting a valid feature configuration from EPL, variant points must be represented in the product line artifacts. Several variant notations have been proposed for use case scenarios, like Product Line Use Case Modeling for Systems and Software Engineering (PLUSS) [11] and Product Line Use Cases (PLUC) [4]. 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* [28] 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 [28]. Here, we consider extra features, such as *Update User Preferences* that, based on the user historical data of searches and purchases, updates user preferences. Figure 1 presents part of the *eShop* feature model, a domain artifact used for representing the common and variant features of a SPL [14, 9, 23]. As a brief introduction about the feature model notation, the relationships between a parent and its child are categorized as: *Optional* (features that might not be select in a specific product), *Mandatory* (feature 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).

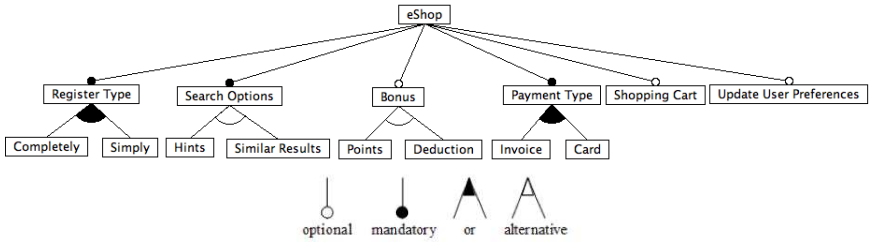

Fig. 1. eShop feature model.

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 12 are never performed together. They are alternative steps: step 1(a) will be present only if the *Shopping Cart* feature is selected (otherwise step 1(b) will be present). 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. Following this approach, it is hard to understand a specific configuration because all possible variants are described in the same artifact. Also, 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 *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 *Search for Products* scenario (the search might also be remote).

Consequently, since the behavior of certain features can be spread among several specifications and each specification might describe several variants, the efforts needed to understand and evolve the product line might increase.

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.

Fig. 2. Buy Products scenarios using the PLUSS notation.

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 3, which also describes the *Buy Products* scenario, denotes a variation point that might assumes the values "*checkout*" or "*buy item*", depending on which product was 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. Another kind of tangling occurs in this case, since segments of the specification are tangled with the variation points. Additionally, SPL members are also described using the same tag notation (see the *Product Definition* section in Figure 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, instead, 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 feature, the inclusion of a new member in the product line might require a deeply review of variation points. Moreover, since the variation points and the product definitions are spread among several scenario specifications, it is hard and time consuming 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 - if the SPL evolved, changes would be propagated throughout many artifacts.

```
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 ...
08 Calculate the shipping costs.
09 Confirm the purchase.
10 Execute the order and sends a request to the Delivery System in ....
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 can...)
VP3 = if (VP0 == P1)
      then ( Requests bonus and payment information.)
      else (Requests payment information.)
VP4 = (Economic, Fast)
VP5 requires (VP0 == P1)
```

Fig. 3. Buy Products scenarios using the PLUC notation.

In summary, both techniques rely on simple composition techniques: filtering variant steps in scenarios or syntactic changes of tag values based on product definition. In this sense, they are similar to conditional compilation techniques, which have been applied to implement variability at source code. Such techniques are not suitable for modularizing the crosscutting nature of certain features, has poor legibility and leads to lower maintainability [3]. As a consequence, we argue that the variability management concern should be separated from the other artifacts and used as a language for generating specific products. The automatic generation of specific product artifacts has being recommended by the SPL community [24, 15, 9], in such way that the combination of generative techniques, aspect oriented programming, and software product line should be further investigated. In this case, in order to support the automatically derivation of product specific artifacts, it is necessary not only a more precise definition of each language used to describe product line artifacts and the variability management concern, but also the weave processes used to combine them. PLUSS and PLUC approaches fail in this direction, since Eriksson et al., although present the metamodel of PLUSS notation [11], do not describe which languages and processes are used for relating use case scenarios to feature models. Likewise, besides Fantechi et al. describe the formal semantics of PLUC [12], this approach does not separate variability management from use case scenarios.

The next section describes our approach that considers scenario variability as a composition of different artifacts. Although in this paper we are focus on

use case scenarios, the idea of separating product line artifacts from variability management is also applied to other SPL views. Moreover, we believe that our modeling framework, which describes the weaving processes for handle variability management, might be customized for design, implementation, and test artifacts.

3 Scenario Variability as Crosscutting

Aiming to represent a clear separation between variability management and scenario specification, and also describe the weave processes required to compose those views, we are proposing a modeling framework that is a customization of the Masuhara and Kiczales (MK) work [25]. The goal of MK framework is to explain how different *aspect-oriented* technologies support crosscutting modularity. Each technology is modeled as a three-part description: the related weave processes take two programs as input, which crosscut each other with respect to the resulting program or computation [25].

Similarly to the MK framework, we represent the semantics of **scenario variability management** as a weaver that takes four specifications as input (*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 4). Combining these input languages, it is possible to represent the kinds of variability that we are interest in: *optional use cases and scenarios*, *quantified changed scenarios*, and *parameterized* scenarios.

A running example of our approach is presented in Section 3.1. After, we describe the semantics of our weaving process. For simplicity, it was decomposed in three subcomponents - 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 [22]. Such decision allows the execution and testing of concise weaving processes descriptions (all related source code is available at SPG web site [2]).

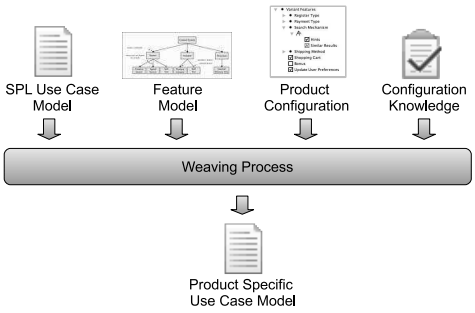

Fig. 4. Overview of the modeling framework.

3.1 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 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.

SPL use case model: defines a set of scenarios that might be used to describe possible variants of the product line. This artifact is not directly concerned with variability management, although some scenarios might be optional, might have parameters, or might change the behavior of other scenarios. Based on the notation proposed by [5], a use case scenario corresponds to a sequence of steps (a tuple of *User Action* x *System State* x *System Response*). Depending on the user action or system state, the behavior of a scenario can be changed. A use case defines a set of scenarios; and a use case model, instead, defines a set of use cases. In this running example, we are considering the following scenarios:

1. **Buy Item Basic Version:** this scenario (Figure 5) specifies the basic behavior of *Buy Products*, assembled in products that are not configured with the *Shopping Cart* and *Bonus* features. It starts from the IDDLE special state (do not extend the behavior of an existing scenario) and finishes at the END of execution (in this case, there is no other behavior to be performed). The clauses *From step* and *To step* are used for describing the possible starting and ending points of execution.

Description: Basic version of Buy Products scenario
From step: IDDLE
To step: END

Id	User Action	System State	System Response
1M	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.
2M	Select the confirm option.		Request payment information.
3M	Fill in the requested information and select the proceed option.		Request the shipping method and address.
4M	Select the <ShipMethod>, fill in the destination address and proceed.		Calculate the shipping costs.
5M	Confirm the purchase.		Execute the order and send a request to the Delivery System to dispatch the products. [RegisterPreference]

Fig. 5. Basic version of Buy Products scenario.

Notice that a parameter *ShipMethod* is referenced in step 4 of Figure 5. 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.

2. **Buy Products with Shopping Cart and Bonus:** this scenario (Figure 6) changes the behavior of the *Buy Products Basic Version* by replacing its first two steps, introducing the specific behavior required by the *Shopping Cart* and *Bonus* features. This scenario also starts from the IDDLE state (*from step* clause), however, it returns to the third step of the *Buy Products Basic Version* (*to step* clause). This behavior is required for products that are configured with *Shopping Cart* and *Bonus* features.

Description: Extended version of Buy Products scenario
From step: IDDLE
To step: 3M

Id	User Action	System State	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 shopping cart.
V2	Select the confirm option.		Request bonus and payment information.

Fig. 6. Extended version of Buy Products scenario.

3. **Search for Products:** this scenario allows the user to search for products. In order to save space, we are only presenting the step (3S) that perform a search based on the input criteria. This step is annotated with the mark [**RegisterPreference**], exposing it as a possible point that the behavior of *Register User Preferences* might starts. The same annotation was used in the *step 5M* of *Buy Products Basic Version*. Such annotations can be referenced in the *from step* and *to step* clauses, reducing the problem of *fragile pointcuts* [6].

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

Id	User Action	System State	System Response
...	...	...	...
3S	Inform the search criteria.		Retrieve the products that satisfy the search criteria. Show a list with the resulting products. [RegisterPreference]

Fig. 7. Search for Products scenario.

4. **Register User Preferences:** this 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**] (see the *from step* clause) annotation and is available in products that are configured with the *Register User Preferences* feature.

```
Description: Register user preferences based on searches and purchases
From step: [RegisterPreference]
To step: END
```

Id	User Action	System State	System Response
1R	-		Update the preferences based on the search results or purchased items.

Fig. 8. Register user preferences based on searches and purchases

Feature model: we have introduced part of this artifact for the eShop product line in Section 2. However, here we are going to present only the features that are fundamental for the running example. Based on the feature model of Figure 9, the *Shopping Cart*, *Bonus* and *Update User Preferences* features are not required; on the other had, the feature *Ship Method* is mandatory and a specific product might be configured with at least one of its child.

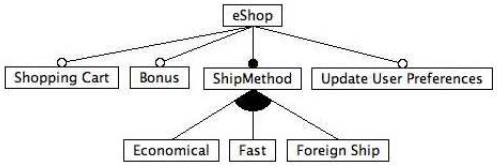

Fig. 9. eShop feature model for the running example.

Product configuration: identifies 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 obeyed the feature model relationships and constraints). Two possible configurations are presented in Figure 10. The first configuration (on the left side of the figure) defines a product that has no support for shopping cart, bonus and updating preferences. Additionally, it supports only the economical and fast ship methods. The second configuration selects all possible features.

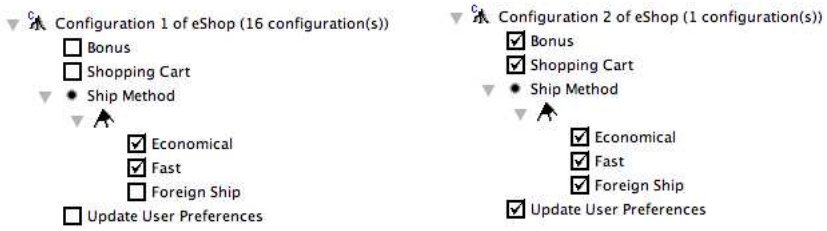

Fig. 10. Examples of product configurations.

Configuration knowledge: relates feature expressions with artifacts that must be assembled in a given product. Such artifact allows, during the product engineering phase, the automatically selection of assets that are required for a specific product configuration. Table 1 presents a configuration knowledge for the running example, enforcing that: if *Shopping Cart* and *Bonus* features are **not** selected, the basic version of *Buy Product* scenario will be assembled; otherwise, the extended version of the same scenario will be assembled; and the *Register User Preferences* scenario will be assembled only if the *Register User Preferences* feature is selected.

Table 1. eShop configuration knowledge for the running example

Expression	**Required Artifacts**
...	...
not (Shopping Cart **and** Bonus)	Basic version of Buy Products scenario
Shopping Cart **and** Bonus	Extended version of Buy Products scenario
Register User Preferences	Register user preferences scenario

Weaving process: After presenting input artifacts for the running example, we are ready to describe the waving process that combine the input languages in order to derive product specific scenarios. In the next section we present, more precisely, the semantics of its components using a low level design. The weave process is composed by four activities, although the last one is optional:

1. **Validation:** 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 is conform to the relationship cardinalities and constraints of the feature model), the process might proceed. Otherwise, an error should be reported. In the running example, both configurations presented in Figure 10 are valid.

2. **Product derivation:** This activity takes as input a (valid) product configuration and a configuration knowledge. Then, each feature expression of the configuration knowledge is checked against the product configuration. If the expression is satisfied, the related scenarios are assembled in the result of this activity. For the running example, Table 2 shows the assembled scenarios for the configurations in Figure 10.

Table 2. Assembled scenarios for each configurations of the running example

Configuration	**Assembled scenarios**
Configuration 1	Basic version of Buy Products
	Search for products
	...
Configuration 2	Extended version of Buy Products
	Search for Products
	Register user Preferences
	...

3. **Scenario composition:** This activity is responsible for composing the scenarios assembled for a specific product configuration. The resulting scenarios of the previous activity, which crosscut each other based on the *from step* and *to step clauses*, are waved. The result is either a use case model with complete paths (all *from step* and *to step* clauses are resolved) or a trace model (a set of all valid sequences of events extracted from the complete paths).
   The complete path is a high level representation, which uses the same constructions of the use case model (scenarios), and can be represented using a graph notation, where each node is labeled with step id. For example, Figure 11 depicts the complete paths for the first and second configurations of our running example. In the left side of the figure, the basic versions of search for product (branch labeled as 1S, 2S, 3S) and buy product (brach labeled as 1M, 2M, ..., 5M) scenarios are presented. Instead, on the right side of the figure, the extended versions of these scenarios are presented. In this case, steps 1M and 2M have been replaced by steps V1 and V2 (because *Shopping Cart* and *Bonus* features are selected) and the step 1R is introduced after steps 5M and 3S (because *Register User Preferences* is selected in this configuration).
   Instead, the trace model can be seen as a low level representation of the use case model. Such notation has a well defined semantic and might be used for model checking and test case generation. Such applications of the trace model are beyond the scope of this paper. More information can be found elsewhere[18, 29, 13]. Here, the trace model is useful for implementing the last activity of our the weave process, binding parameters, and represents all possible sequences of events in a specific product configuration.

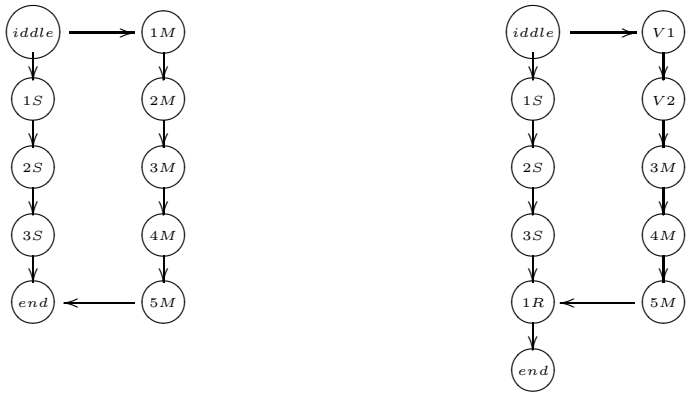

Fig. 11. Complete paths for the first (left) and second (right) configurations.

For example, the trace model for the first configuration is the set of sequences:

$$\begin{aligned} Trace_{C1} = \{&<>, < iddle >, < iddle, 1S >, < iddle, 1S, 2S >, \\ &< iddle, 1S, 2S, 3S >, < iddle, 1S, 2S, 3S, END >, \\ &< iddle, 1M >, < iddle, 1M, 2M >, \ldots, \\ &< iddle, 1M, 2M, 3M, 4M, 5M >\} \end{aligned}$$

4. **Binding of parameters:** This optional activity takes as input the assembled scenarios and the product configuration and generates a trace model resolving all parameters referenced by the steps. For example, step 4M in Figure 5 has a reference to the parameter *ShipMethod*. The parameter values are defined in the product configuration. For instance, in the first configuration (Figure 10), the parameter *ShipMethod* might assume the values *Economical* or *Fast*. In order to reduce the coupling between scenario specifications and feature model, a mapping (or environment) is used for relating parameter to features. For each trace that contains a parameterized event (or step), this activity creates a new trace for all of the possible parameter values. Consequently, the traces $< iddle, 1M, 2M, 3M, 4M >$ and $< iddle, 1M, 2M, 3M, 4M, 5M >$, will generate the following sequences:

$$\begin{aligned} &< iddle, 1M, 2M, 3M, 4M.Economical >, \\ &< iddle, 1M, 2M, 3M, 4M.Fast >, \\ &< iddle, 1M, 2M, 3M, 4M.Economical, 5M >, \\ &< iddle, 1M, 2M, 3M, 4M.Fast, 5M > \end{aligned}$$

Next, we introduce the modeling framework used to formally describe the composition processes introduced in this running example.

3.2 Modeling Framework

As presented before, we are proposing, based on the Masuhara and Kiczales work [25], a crosscutting variability management approach. Our weaving process is composed by three components, formally described in sections 3.3, 3.4, 3.5. Actually, each of this components is also a weaver. The modeling framework used to describe these wavers is an 8-tuple (Eq. 1 and Table 3), which highlights the role of each language in the composition process.

$$T = \{o, o_{jp}, L, L_{id}, L_{eff}, L_{mod}, p, meta\}, where: \quad (1)$$

Table 3. Modeling framework elements.

Element	Description
o	Output language used for describing the results of the weave process
o_{jp}	Constructions used for representing the output language join points
L	Set of languages used for describing the input specifications
L_{ID}	Constructions in each input language used for identifying join points
L_{EFF}	Effect of l' constructions in the weaving process, where $l \in L$
L_{MOD}	Modular unities of each input language
p	Weave process representation
$meta$	An optional argument used for customizing the weave process

For each weaver, we describe the input languages (L) that crosscut each other in order to generate a representation in the output language (o). Differently from the MK work, we have included a low level description of the weave process (the p element of the tuple) in our variability mechanism framework.

3.3 Product derivation weaver

This weaver is responsible for the first two activities of our variability management approach: validate a product configuration against a feature model and, based on a configuration knowledge, select a subset of the use case model. So, this weaver takes as input a *feature model*, a *product configuration*, and a *configuration knowledge*. The result is a set of scenarios that are valid for the product configuration.

A product configuration has a similar structure to the feature model: one feature representing the root element of the configuration. Each feature, instead, can have multiple children. In feature modeling, its is also necessary to represent the feature id, name, classification (optional, mandatory, alternative), and

properties. Listing 1.1 presents the abstract descriptions of feature models and product configurations[1].

Listing 1.1. Product Configuration

```
1 -- Other types are not presented here
2 type Root = Feature
3 type Children = [Feature] -- a list of features
4 data Feature = Feature Id Name Type Children Props
5 data FeatureModel = FeatureModel Root
6 data ProductConfiguration = ProductConfiguration Root
```

A *configuration knowledge* corresponds to a set of the pairs (*feature expression,related artifacts*). Each pair is named here as a *configuration*. Given a *product configuration* (PC) and a valid *configuration knowledge* (CK), we need to evaluate each feature expression in order to identify if the product satisfies it. If a feature expression is satisfied, the related artifacts will be present in the final product. A feature expression can be either a reference to a feature (**feature id**) or a composite expression (**and expression**, **or expression**, or **not expression**). Listing 1.2 presents the *configuration knowledge* abstract representation in Haskell.

Listing 1.2. Configuration Knowledge

```
1 type Exp = FeatureExpression
2 type Configuration = (Exp, ArtifactList)
3 type ConfigurationList = [Configuration]
4 data FeatureExpression =
5  FeatureRef Id |
6  NotExpression Exp |
7  AndExpression Exp Exp |
8  OrExpression Exp Exp
9 data ConfigurationKnowledge = ConfigurationKnowledge ConfigurationList
```

The *pdWeaver* function (lines 4-7 in Listing 1.3) implements this weaver. It takes as input a *feature model* (FM), a *product configuration* (PC), and a *configuration knowledge* (CK); and produces the set of selected artifacts (in this case, a set of scenarios) if the product is a valid instance of the feature model - otherwise the *InvalidProduct* error is thrown. For each configuration (x) in the configuration knowledge, an evaluation (line 7) of its expression is performed. If the expression is evaluated as *True* (the *product* satisfies the feature expression), the related scenarios will be included in the resulting list of artifacts. Finally, the *expression* (line 12) and *artifacts* (line 13) auxiliary functions returns, respectively, the expression and artifacts of the configuration that is in the head of the list *x:xs* (see the definition of a single configuration in Listing 1.2).

[1] In the Haskell programming language, the *type* reserved word is used for defining a type synonymous. On the other hand, the *data* reserved word is used for defining a new data type.

Listing 1.3. The *configure weaver* function

```
type PC = ProductConfiguration
type CK = ConfigurationKnowledge

pdWeaver :: FM -> PC -> CK -> ArtifactList
pdWeaver fm pc ck =
 if not (validInstance fm pc) then error InvalidProduct
 else configure pc ck

configure :: PC -> CK -> ArtifactList
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)
```

Now, we can summarize this weaver using our customized framework. The result of this composition is the set of all valid scenarios for a given product configuration. Thus, our output language (o) is the use case model, since it can describe a set of scenarios. Additionally, each scenario definition is a *joinpoint* (o_{jp}) in the use case model, allowing its selection. Feature models (FM), product configurations (PC) and the configuration knowledge (CK) are used as input of the weaving process (L={FM, PC, CK}). The selected features correspond to the pointcut constructions in the product configuration (PC_{ID}); and its effect is to define a member of the software product line (PC_{EFF}). Each feature expression is used as a pointcut in the configuration knowledge (CK_{ID}), and its effect (CK_{EFF}) is to relate a feature configuration with a set of artifacts (in our case, only scenarios). The weaver (p) is the *pdWeaver* function defined in the Listing 1.3. Finally, each selected feature is a modular unit in the product configuration (PC_{MOD}), since it can be reusable in other product configurations. On the other hand, the configuration knowledge modular unities (CK_{MOD}) are the feature expressions and configurations.

3.4 Scenario composition waver

This weaver is responsible for the third activity of our variability management approach. It aims to compose variant scenarios of a use case [5] and is applied whenever a use case scenario supports different paths of execution, based on the product configuration.

This mechanism takes as input the selected use case model (a set of scenarios) of a specific SPL member. Each variant flow (usually a partial specification) must be composed in order to generate a concrete scenario. A variant scenario might refer to steps either in basic or other alternative scenarios. In order to compute the complete paths defined by a scenario, we need to compose, recursively, the events that precede all of the steps referenced by its *from step clause* (until the IDDLE state), plus its own steps, plus all the events that follow all of the steps referenced by its *to step clause* (until the END state)[2].

[2] IDDLE and END states are predefined steps that represent the *beginning* and the *end points* of a specification.

Listing 1.4 presents the abstract representation of the use case model, use case, and scenario artifacts. Briefly, a use case model has a name and a set of use cases. A use case has an id, a name, a description, and a list of scenarios (that defines its basic, alternative, and exception flows). A scenario has an id, a description, a *from step clause* (a list of references for existing steps), a list of steps, and a *to step clause* (also a list of references for existing steps). A step has an id, a reference to the parent scenario, a specification in the form of a tuple (user-action x system-state x sytem-response), and a list of annotations that can be used to semantically identify the step (avoiding the problem of fragile pointcuts - see Section 3.1). Finally, a reference to a step can be either a *reference to a step id* or *a reference to a step annotation*.

Listing 1.4. Use Case and Scenario representation

```
1 type Detail = (Action, State, Response)
2 type FromStep = [StepRef]
3 type ToStep = [StepRef]
4 data TraceModel = [EventList]
5 data UseCaseModel = UseCaseModel Name [UseCase]
6 data UseCase = UseCase Id Name Description [Scenario]
7 data Scenario = Scenario Id Description FromStep [Step] ToStep
8 data Step = Step Id Scenario Detail [Annotation]
9 data StepRef = IdRef Id | AnnotationRef String
```

This weaver can be configured (the *META* element of our modeling framework) to return: a) a product specific use case model, contemplating all complete paths of each scenario; b) a trace model (see Section 3.1) without parameters, computed for each complete path of the resulting scenarios; or c) a trace model, with resolved parameters, for each complete path of the resulting scenarios. The *binding parameter* weaver is described in the next section.

The *completePaths* function (lines 1-4 in Listing 1.5) takes as input the whole use case model (*ucm*) and a scenario (*scn*); and returns all complete paths (a list of *step lists*) of *scn*. The function fromList (called in line 3) is used to compose all complete paths extracted from the *from step clause*. In a similar way, the function *toList* (called in line 4) is used to compose all complete paths extracted from the *to step clause*. The *match* function (also called in lines 3 and 4), retrieves all the steps in *ucm* that satisfy all *step references* in *from step* or *to step* clauses. Currently, this matching is based on the *step id* (a syntactically reference) or on the list of *step annotations* (a semantic reference). The "+++" operator denotes a distributed list concatenation.

The *traceModel* function (lines 7 and 8 of Listing 1.5) computes a set of all valid sequences of events from a list of steps. Such function take as argument a function *f* and the list of steps (*x:xs*) (it is possible to call this function passing all *completePaths* from a given step). Currently, the valid functions *f*, which can be passed as the first argument, are the *stepId* function (returns the id of a given step); and the *bind e x* function (computes the parameter values of a given *step* in a specific *feature environment*) In the next section we present, in more details, the *bind* and *feature environment* constructions.

Listing 1.5. The *completePaths* and *traceModel weaver* functions

```
completePaths :: UseCaseModel -> Scenario -> [StepList]
completePaths ucm scn =
 (fromList ucm (match ucm (fromStepsOf scn)) +++ [stepsOf scn]) +++
 (toList ucm (match ucm (toStepsOf scn)))

-- called as traceModel stepId (x:xs) or traceModel (bind e) (x:xs)
traceModel f [] = [[]]
traceModel f (x:xs) = [] : (f x) ^ (traceModel f (xs))
```

Here we instantiate our modeling framework in order to describe this weaver. First of all, the output languages can be either a set of *weaved* scenarios (or a use case model) or a computed trace model (in this case, $o = (UseCaseModel \mid TraceModel)$). The use case model *joinpoints* (o_{jp}) are composed by the scenario and step definitions. On the other hand, the *joinpoints* for the trace model are composed by each valid sequence of events (an element of a trace model).

The input language is a set of scenarios (or a use case model) that refer to each other. The pointcut (L_{ID}) are specified by the *from steps* and *to steps* clauses of each scenario. The effect of a scenario definition (L_{EFF}) corresponds to the requirement specification of valid sequence of events, essencial for product development and testing. Additionally, the scenarios are also the modular unities.

The *completePaths* and *traceModel* (Listing 1.5) functions are the weaving processes ($P = (completePaths \mid traceModel)$) of this variability mechanism. Finally, as we have explained before, such mechanism is parameterized (the $META$ element of our customized framework) in order to select which kind of representation will be generated (composed use case model or trace model); and, once the trace model representation was selected, identify if the parameters will be resolved or not.

3.5 Bind parameters weaver

Parameters are used in scenario documents in order to create reusable requirement specifications. This kind of variability can be applied whenever two or more scenarios share the same behavior (the sequence of actions) and differ in relation only to values of a same concept. For instance, Figure 5 depicts the *Buy Products* scenario that can be reused for different *ship methods*. Without this parameterized specification, and aiming at automatically generate a test case suite (for example) with a good coverage, it would be necessary to create a specification for each kind of method.

This weaver takes as input a parameterized *scenario specification* and a *product configuration*, which defines the domain values of a parameter. Thus, in order to reduce the coupling between a scenario parameter and a feature, we are proposing an environment that relates them. Features related to parameters must be either an **alternative feature** or an **or feature** [14, 10, 9].

The implementation of this weaver consists in a call to the *traceModel* function (Listing 1.5) with the *bind e* partial function as first parameter. The *bind* function (lines 1-5 of Listing 1.6) takes as input an environment (e), that maps a parameter into a feature, and a step (s). Then, it extracts all parameters from s, and returns a *String* representation with the corresponding parameter values. Each text between the symbols "<" and ">" (defined in the user action, system state, or system response of a step) is treated as a parameter and must be defined in the environment (otherwise, an type exception is thrown).

Listing 1.6. The *bind waver* function

```
1 bind :: Environment Feature -> Step -> String
2 bind e x =
3  if (length (extractParameters (details x)) == 0)
4   then stepId x
5   else stepId x ++ (extractParameterValues e x)
```

The *extractParameterValues* function (called at line 5 of Listing 1.6) is responsible for extracting the related parameter values from a feature. Also, each parameter must be related with an **alternative feature** or an **or feature** present in a product configuration.

Instantiating the *bind parameter weaver* to our modeling framework, we identify that the trace model is the resulting language (o) and its set of joinpoints (o_{jp}) is composed by each *communicated data* present in the trace of events. Also, the mechanism takes as input a set of scenarios and a *product configuration*. Steps defined in the set of scenarios can refer to parameters declared in a *feature environment*, that relates each parameter id with a feature in the product configuration. Thus, the pointcut constructions are, respectively, the parameters defined in the scenario specifications and the **alternative** and **or features** selected in the product configuration. The weaving process (p) is the composition $f \circ g$, where f is the *traceModel* function (Listing 1.5) and g is the *bind* function (Listing 1.6).

4 Comparative Analysis

In this section, we compare our approach with three related approaches, briefly introduced in the next section. After, we describe our analysis criteria and present the comparative results.

4.1 Overview of related approaches used in this comparison

We have selected three works as basis for our comparative analysis, including: *Featuring Reuse-Driven Software Engineering Business*(FeatuRSEB) [16], *Product Line Use case modeling for System and Software engineering* (PLUSS) [11], and *Product Line Use Cases* (PLUC) [4].

These works share the same characteristic of combining variability management with use case models in order to describe scenario variabilities. Additionally, they are representative in the sense of supporting mechanisms for representing variabilities. Next, we present a brief description of each technique. More details about the PLUSS and PLUC notations are presented in Section 2.

- **FeatuRSEB:** Approach that joins ideas from a domain driven process (Feature Oriented Domain Analysis [23]) with a reuse driven process (Reuse-Driven Software Engineering [20]). One of the main goals of FeatuRSEB work is to integrate feature models with use cases.
- **PLUC:** An extension for use case modeling that allows variability representation using tags in scenario specifications. A central characteristic of this work is that variation points are implicitly enclosed into use case specifications. A formal description of the PLUC composition process is presented in [12].
- **PLUSS:** Based on FeatuRSEB, this approach extend the support for different kinds of requirement variabilities, enabling the relationship between features and use cases, scenarios, and steps. This approach also describes an integrated metamodel for use cases an scenarios.

4.2 Results

Based on three selected criteria to perform the analysis, we have evaluated our work and each of the approaches briefly described in the previous section. The chosen criteria set allow us to verify the benefits of our work considering: the kinds of variability supported; the separation between the role of each language in scenario specification and variability management; and the design expressiveness of each mechanism.

Variability Support Criterion: This criterion is related to the support of a given technique for describing the classes of variability (optional use cases and scenarios, changed scenarios, crosscutting requirements, and parameterized scenario) presented in [11].

The FeatuRSEB approach allows the relationship between features and use cases, thus there is support for optional use case variability. However, in order to represent an optional scenario, it must be defined as an use case extension. If the number of optional scenarios is great, the resulting use case decomposition could be not suitable for development activities. Changed scenarios and crosscutting variability are well supported by alternative and extension flow specifications. However, only support for syntactic composition is present. Parameterized variability is also supported by FeatuRSEB.

In PLUC, use cases are classified as *product line* or *product specific* use cases. A *product line use case* is present in all members of the family. A *product specific use case*, instead, is presented only in the required members. Thus, this approach has support for optional use case variability, although there is no explicit support

for describing optional scenarios. Additionally, each scenario variability must be enclosed in the same use case, implying that a scenario can not change different use cases (no support for crosscutting variability). Changed scenarios are described as alternative scenarios. Parameterized variability is supported in PLUC by tags that can be used in the entire use case specification. The domain values of each tag is also described in use case document.

The PLUSS approach allows the relationship between a feature, presented in a feature model, and use cases, scenarios, and steps. Thus, this approach support both optional use cases and scenario variabilities. In PLUSS, each scenario specification is responsible for representing all of its variants, in such way that the changing scenario variability is described relating optional steps with features. Parameterized scenario variability is also supported, however there is no explicit mechanism to represent that a scenario can change the behavior of different scenarios (so, the crosscutting variability is not supported).

Our approach supports optional use case and scenario variabilities by relating them with feature models (the *product derivation weaver* is responsible for this kind of variability). Such relationship is preserved in a third artifact, named as product configuration. In our work, the support for changing scenario variability uses variant scenario specifications, which can start and finish at different steps of a use case model (the *scenario composition weaver* is responsible for this kind of variability). The references to these steps can be either syntactic (using the step id as reference) or semantic (using one annotation as reference). Parameters are allowed only in step details (user-action, system-state, system-response), and their domain values are defined in the product configuration (the *parameterized weaver* is responsible for this kind of variability).

Table 4 summarizes the comparative analysis based on this criterion. A "+" mark represents a good support for variability; a "-" mark is used to represent a low variability support; and a mark absence represents that an approach has no support for a kind of variability.

Table 4. Variability support resulting analysis

Kind of Variability	**FeatuRSEB**	**PLUC**	**PLUSS**	**Our approach**
Optional use cases	+	+	+	+
Optional scenarios	-		+	+
Changing scenarios	+	-	-	+
Crosscutting	+			+
Parameterization	+	+	+	+

Separation of Concerns (SoC) Criterion This criterion identifies if a given technique presents a clear separation between variability management and scenario specification.

Applying the use case extension mechanism for representing scenario variability is the major SoC problem in the FeatuRSEB approach. Such mechanism was proposed for describing an optional and reusable behavior in the context of a single product development. Adapting this mechanism to represent optional scenarios in a software product line context can result in the undesired explosion of use cases.

PLUC defines tags in the scenario specification in order to describe variabilities. The domain values of each tag can represent: a member of the product line or an parameter value/optional behavior that must be selected based on a specific product. All tags are enclosed into the use cases. In this way, a use case specification is responsible for describing a set of scenarios, the relevant product instances that affect its behavior, and the domain values of each tag. Some of the tags are conceptually related with optional features, that might affect several artifacts. Thus, a change in the feature model can affect tag definitions in several use cases. The same problem might occur with the inclusion of a new member in the product family.

In the PLUSS approach, the changing scenario variability is modeled by relating each step variation with a feature of any type in the feature model. A single scenario specification is responsible for representing all of its variants. As result, the behavior of a same feature can be spread in different points of a use case model. Also, there is no clear separation between the scenario specification and the implemented variability mechanism, being difficult to reason about a feature in a isolated way. Finally, changes are not localized: if the feature needs to be changed, multiple points in the same scenario (or in different scenarios) could be reviewed.

One of the goals of our approach is to define a clear separation of the role of each language. Thus, a scenario specification is used only to describe a sequence of events. Similarly, a feature model is responsible only for defining the common and variant features of a SPL. Another artifact, the product configuration, is responsible for relating features with software artifacts. Such kind of SoC offers greater levels of modularity, enabling: a) changes to be performed in an isolated way; and b) better comprehensibility of each artifact.

Design Expressiveness Criterion This criterion aims to identify how clear is the design of each compared work. It is important to verify such expressiveness because recent approaches for SPL development follow a model driven thought, in which the semantics of each model and the transformation rules among them are necessary.

In the FeatuRSEB work, the *optional use case* variability is supported by the relationship between a use case model and a feature model. However, the approach does not describe **how** the use case model and the feature model are related. Additionally, the mechanism used to weave scenario specifications (changing scenarios) is only implicitly described in RSEB [20]. After FeatuRSEB was proposed, Jacobson et al. presented a better description of use case extension flow semantics [21].

The PLUSS approach presents a meta-model responsible for describing the relationship between use cases (or a scenarios) and features. According with the meta-model cardinalities, we can not describe a relationship that defines two or more features as being necessary to configure a SPL member with a specific artifact (use case or scenario). This is a common example in SPL configuration. Another question related with the semantics of the PLUSS approach is the use of parameters to represent crosscutting variabilities, instead of quantified relationships among scenarios. We argue that parameters and crosscutting scenarios are two orthogonal kinds of variability.

In the previous works, there is a lack of design definition, which is necessary for a better understanding of the variability management mechanisms. The result is a gapping for introducing the previous works in a *generative* driven SPL. Nowadays, generative techniques are applied to derive and select software components [9, **?**,**?**,24]. However, with no precise semantics to requirement variability management, few efforts were applied to automatically generate member specific requirements of a SPL. Our work, differently, presents a low level design of each variability mechanism. In order to increase our confidence, we chose to represent the variability weaving processes using a programming language[3]. In this way, we have contributed to reduce the gap for applying an automatically derivation of SPL requirement product specification.

5 Related Work

Our work is linked to the body of research related to SPL variability management, crosscutting modeling, and use case scenario composition. This section details some of these approaches, relating them to the proposed solution.

5.1 Variability Management

Pohl et al. argue that variability management should not be integrated into existing models [27]. Their proposed Orthogonal Variability Model (OVM) describes traceability links between variation points and the conceptual models of a SPL. Our approach can be applied in conjunction with OVM, since it also decouple variability representation and offers a crosscutting mechanism for specific product requirement derivation.

Hunt and McGregor proposed a pattern language for implementing variation points [19]. The goal is to create a catalogue that relates patterns of variabilities with good alternatives for implementing them. It is not the goal of our work to explicitly relate pattern variabilities with one of the proposed mechanisms. However, our framework is also a language for representing requirement variability mechanisms. Additionally, we believe that it can be customized to describe mechanisms in other SPL models.

[3] As we said before, the source code is available at the topic Software and Processes of the SPG site [2].

5.2 Crosscutting Modeling

As result of the convergence between model driven development (MDD) and aspect oriented software development (AOSD), several works were proposed in order to represent weaving mechanisms using abstract state machine and activity diagrams [26, 8]. Our work, on the other hand, describes weaving mechanisms for scenario specification using a functional notation. First of all, we believe that the textual language is the preferred representation for scenario specification. Second, the use of a functional notation resulted in a more concise model of the weaving mechanisms.

The AMPLE project aims to combine ideas from MDD and AOSD, in order to bind the variation points at the different SPL models [1]. Our work is closed related with the AMPLE objectives, since we describe the representation (meta-model) and relationships between the languages involved in requirement variability and how they crosscut each other to derive a SPL member specification.

5.3 Use Case Scenario Composition

In order to avoid the problem of *fragile point-cuts*, Rashid et al. proposed a semantic approach for scenario composition [6]. Such approach is based on natural language processing. Using our quantified changing mechanism (Section **??**), a scenario composition can be represented using references to step id or step annotations, which also reduce the problem of *fragile point-cut*. However, introducing the Rashid's approach in our environment could be implemented without break our quantified changing mechanism. It would be only necessary to extend the *match* function definition, called at line 3 of Listing 1.5.

6 Conclusions and Future Work

In this paper we argued about the crosscutting nature of variability management. Since a variant feature might require variant points to be spread in many artifacts, it does not make sense to tangle variability management with the other product line artifacts.

In order to separate variability management and scenario specification, we presented a crosscutting modeling framework for representing scenario variabilities. Our approach covers two important criteria previously defined: supporting for different kinds of variabilities and separation of concern (SoC) between variability representation and scenario specification. Although our modeling framework was proposed for handle scenario variabilities, we believe that it could be customized to be applied in other SPL artifacts. Particularly, optional and parameterized artifacts are also relevant for non-functional requirements in a product line.

As future work, we want to identify the relationship between variabilities at different SPL artifacts (in a first moment, test artifacts). Such kind of relationships can help in the product generation and traceability activities of a SPL.

The modeling framework proposed in this work takes a step in this direction, since the composition process used to derive product specific scenarios have been already implemented.

References

1. Ample project, online: http://ample.holos.pt/ (2007).
2. Software productivity group, online: http://www.cin.ufpe.br/spg (2007).
3. 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 *Fifth International Conference on Generative Programming and Component Engineerig (GPCE'06)*, Portland, USA, 2006.
4. A. Bertolino and S. Gnesi. Use case-based testing of product lines. In *ESEC/FSE'03*, pages 355–358, New York, NY, USA, 2003. ACM Press.
5. G. Cabral and A. Sampaio. Formal specification generation from requirement documents. In *Brazilian Symposium on Formal Methods, SBMF'06*, Natal, Brazil, sep 2006.
6. R. Chitchyan, A. Rashid, P. Rayson, and R. Waters. Semantics-based composition for aspect-oriented requirements engineering. In *AOSD '07*, pages 36–48, New York, NY, USA, 2007. ACM Press.
7. P. Clements and L. Northrop. *Software product lines: practices and patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
8. T. Cottenier, A. van den Berg, and T. Elrad. Model weaving: Bridging the divide between elaborationists and translationists. In *Aspect Oriented Modelling(AOM'06)*, Genova, Italy, 2006.
9. K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional, 2000.
10. K. Czarnecki and C. H. P. Kim. Cardinality-based feature modeling and constraints: A progress report. In *International Workshop on Software Factories (online proceedings)*, 2005.
11. M. Eriksson, J. Borstler, and K. Borg. The pluss approach - domain modeling with features, use cases and use case realizations. In *SPLC'05*, 2005.
12. A. Fantechi, S. Gnesi, G. Lami, and E. Nesti. A methodology for the derivation and verification of use cases for product lines. In *SPLC*, 2004.
13. C. Feitosa and A. Mota. Unifying csp models of test cases and requirements. In *Brazilian Symposium on Formal Methods, SBMF'06*, Natal, Brazil, sep 2006.
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. J. Greenfield, K. Short, S. Cook, and S. Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, 2004.
16. M. Griss, J. Favaro, and M. d' Alessandro. Integrating feature modeling with the rseb. *ICSR'98*, 1998.
17. 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.
18. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
19. J. Hunt and J. McGregor. Implementing a variation point: A pattern language. In *Variability Management Workshop, SPLC'06*, 2006.

20. 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.
21. I. Jacobson and P.-W. Ng. *Aspect-Oriented Software Development with Use Cases*. Addison-Wesley Professional, 2004.
22. P. Jones. *Haskell 98 Language and Libraries: the Revised Report*. Cambridge University Press, 2003.
23. K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature oriented domain analysis: feasibility study, 1990.
24. C. W. Krueger. New methods in software product line practice. *Commun. ACM*, 49(12):37–40, 2006.
25. H. Masuhara and G. Kiczales. Aspect-oriented programming. In *ECOOP: Proceedings of the European Conference on Object-Oriented Programming*, 2003.
26. N. Noda and T. Kishi. An aspect-oriented modeling mechanism based on state diagrams. In *AOM'06*, Genova, Italy, 2006.
27. K. Phol and G. B. ands Frank Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.
28. K. Pohl and A. Metzger. The eshop product line. online: http://www.sei.cmu.edu/splc2006/eShop.pdf.
29. B. Roscoe. *The theory and practice of concurrency*. Prentice-Hall, 1998.