Modeling Scenario Variability as Crosscutting Mechanisms

Rodrigo Bonifácio
Informatics Center
Federal University of Pernambuco
Recife, Brazil
rba2@cin.ufpe.br

Paulo Borba
Informatics Center
Federal University of Pernambuco
Recife, Brazil
phmb@cin.ufpe.br

ABSTRACT

Variability management is a common challenge for Software Product Line (SPL) adoption, since developers need suitable mechanisms for specifying and implementing variability that occurs at different SPL artifacts (requirements, design, implementation, and test). In this paper, we present a novel approach for use case scenario variability management, enabling a 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 separate 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 each specific SPL member. After applying our approach to different case studies, we achieved a better feature modularity and scenario cohesion.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements—*Languages, Methodologies*; D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Design, Documentation

Keywords

Software product line, variability management, requirements models

1. INTRODUCTION

The support for variation points enables product customization from a set of reusable assets [23]. However, variability management, due to its inherent crosscutting nature, is a common challenge in software product line (SPL) adoption [10, 23]. First, nontrivial features often require associated variation points to be scattered through SPL artifacts. Second, some approaches include product variant and configuration information tangled with software engineering artifacts. Both problems can be observed for use case scenario specifications.

Several authors have proposed the use of *aspect-oriented* mechanisms to better modularize the specification of crosscutting concerns [21, 9]. These techniques minimizes the first problem, since they can be used to modularize the specification of certain features. However, they do not support the different sources of variability that occur in SPL requirements. With respect to the second problem, existing approaches [7, 13] proposed to scenario variability management do not offer a clear separation between variability management and scenario specification. As a consequence, in the case where details about product variants are tangled with use case scenarios, the removal of one variant from the product line requires changes to all related scenarios. In summary, it is difficult to evolve both representations.

So in this paper we go beyond the common-variant scenario composition issues and consider a more encompassing notion of variability management, including artifacts such as feature models [15, 11] and configuration knowledge [11, 23]. We explain this as a crosscutting phenomenon, using Masuhara and Kiczales work [20], and apply this view of *variability management as crosscutting* for modularizing SPL use case scenarios, providing the necessary separation between variability and scenario specification concerns. We also formalize the derivation of product specific scenarios in our approach, as demanded by current SPL generative practices [19]. This formalization is based on our framework for modeling the composition process of scenario variabilities with feature models, product configuration, and configuration knowledge. Besides supporting the mentioned separation of concerns, this framework helps to precisely specify how to weave the different representations in order to generate specific scenarios for a SPL member. Therefore, the main contributions of this work are the following:

- Characterization of the languages of variability management as a crosscutting concern and, in this way, proposing an approach where variability concerns are separated from other concerns (Section 3).
- A framework for modeling the composition process of scenario variability mechanisms (Section 4). This framework gives a basis for describing variability as

AOSD'09, March 2–6, 2009, Charlottesville, Virginia, USA.

crosscutting mechanisms; but, differently from existing works [22, 16], it considers the contribution of different input languages: feature models, product configuration, configuration knowledge and SPL use case model. Moreover, the reference implementation provided for each variability mechanism corresponds to the essential parts of a tool environment for scenario variability management. In this paper we focus just in these essentials.

- Specification of three sources of variability for use case scenarios: variability in function (Section 4.1), variability in data (Section 4.2), and variability in control flow (Section 4.3). This specification provides a more formal semantic representation when compared to existing works; which is an important property for supporting the automatic derivation of product specific artifacts. Although the sources of variability presented here are not complete, we believe that our modeling framework is able to represent other interesting ones, such as context-aware adaptability.

Since our concept of crosscutting mechanism is based on the Masuhara and Kiczales work [20], a smaller contribution of this paper is to 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 that involves different input specifications that contribute to derive a specific member of a given SPL.

Finally, we evaluate our approach (Section 5) by comparing it to alternative approaches using different product lines. We also relate our work with other research topics (Section 6) and present our concluding remarks (Section 7). In the remainder of this paper, we use the word *scenario* as a synonym for the textual specifications of use cases.

2. MOTIVATING EXAMPLE

In order to customize specific products, by selecting a valid feature configuration, variation points must be represented in the product line artifacts. Several notations for representing variation points in use case scenarios have been proposed, such as Product Line Use Cases (PLUC) [7] and Product Line Use Case Modeling for Systems and Software Engineering (PLUSS) [13]. However, in spite of the benefits of variability representation, existing approaches 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* [24] as a motivating example.

The primary 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, corresponding to a product family composed of 72 valid configurations [24]. In this paper we consider additional features and use cases, such as *Update User Preferences*, which updates the user's preferences based on her historical data of searches and purchases.

Figure 1 presents part of the EPL feature model [15, 11], which represents the common and variable features of our example. Here we represent it by a tree like notation where relationships between a parent feature and its children are categorized as *Optional* (features that might not be selected in a specific product), *Mandatory* (feature that must be selected, if the parent is also selected), *Inclusive Or* (one or more sub-features might be selected), and *Exclusive Or* (exactly one sub-feature must be selected for each product). Besides these relationships, feature models allow the specification of constraints among features. For instance, the constraint (*Shopping Cart* $\Leftrightarrow$ *Bonus*) states that the feature Shopping Cart is selected iff the feature Bonus is also selected.

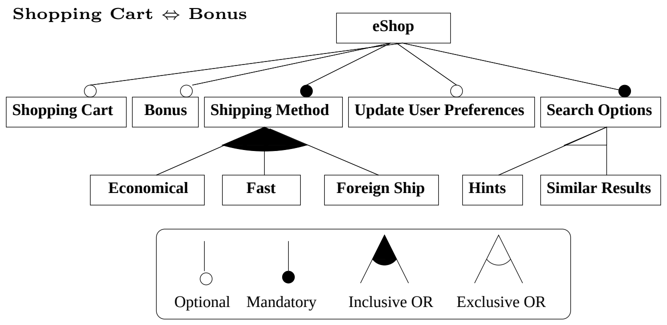


Figure 1: eShop feature model.

Figure 2 depicts the *Buy Product* scenarios written in the PLUSS notation. Notice that a single artifact is used to represent all valid configurations related to this scenario, mixing common behavior, variant behavior, and configuration information (feature selection inside square brackets). For example, steps 1(a) and 1(b) 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 whether the *Bonus* feature is selected or not). Finally, Step 6 is optional and would be present only if the feature *Update User Preference* was selected.

As a consequence, since all possible variants are described in the same artifact, the behavior of specific products is difficult to understand with the PLUSS approach. In addition, this tangling between *common* and *variant* behavior results in maintainability issues: introducing a new product variant requires changes in several points of existing scenarios. For example, including a *B2B Integration* feature, which allows the integration between partners in order to share their warehouses, changes 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). The effort needed to understand and evolve a product line increases, since the specification of certain features is scattered through several scenarios, and each scenario describes several configurations.

Id	User Action	System Response
1(a)	Select the checkout option. [ShoppingCart]	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. [not ShoppingCart]	Present the selected product. The user can change the quantity of items that he wants to buy. Calculate and show the amount to be paid.
2(a)	Select the confirm option. [Bonus]	Request bonus and payment information.
2(b)	Select the confirm option. [not Bonus]	Request payment information.
3	Fill in the requested information and select the proceed option.	Request the shipping method and address.
4	Select the $Shipping-Method$, 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 session option. [Update User Preferences]	Register the user preferences.

Figure 2: Buy Products scenarios using PLUSS.

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:

```
P = (P1, P2)
```

Variation points:

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

Figure 3: Buy Products scenarios using PLUC.

Differently, PLUC introduces special tags for representing variation points 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 assume the values "*checkout*" or "*buy product*", depending on which product is being 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 a scenario specification.

Another kind of tangling occurs in this case. The specifications of common and variant behavior are separate, but both are tangled with the variation points. Additionally, the definition of the SPL members, described using the same tag notation, is scattered throughout the *Products Definition* section of many scenarios (Figure 3). Indeed, differently from PLUSS, 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 configured by an implicit selection of the *Shopping Cart*, *Bonus*, and *Update User Preferences* features; in contrast to the second product that is not configured with these 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 the scenarios' *Variation Points section*. This problem does not occur in the PLUSS notation. Moreover, due to the variation points and the product definitions are spread among several scenario specifications, it is hard and time consuming to keep consistent the relationships between them. Finally, the same definitions (product configuration and variation points) are often useful to manage variabilities in other artifacts, such as design and source code, implying that PLUC requires the replication of such definitions in different SPL views.

In conclusion, both PLUSS and PLUC do not present a clear separation between variability assets and scenario specifications, which compromises the evolution of a SPL.

3. SVCM APPROACH

To solve the problems mentioned in the previous section, we introduce now our approach for modeling variabilities in use case scenarios, which was named *Modeling Scenario Variability as Crosscutting Mechanisms* (MSVCM). It improves the separation of concerns between variability management and scenario specifications, dealing with scenario variability as a composition of different artifacts: SPL use case model, feature model, product configuration, and configuration knowledge.

Motivated by the Masuhara and Kiczales (MK) work [20], our approach for **scenario variability management** relies on a weaving process that takes as input the aforementioned artifacts, which crosscut each other with respect to the resulting product specific use case model (Figure 4). The semantics of the weaving process (and the meta-model of the input and output languages) are described using the Haskell programming language. This led to concise descriptions and kept our model close to the MK work, where their weaving processes are specified using Scheme— another functional programming language. The related source code is available at a web site [2]

In what follows, we detail our approach showing how it can be used to specify the motivating example. Several artifacts of each input model are shown; mentioning the contribution of these models to the whole weaving process.

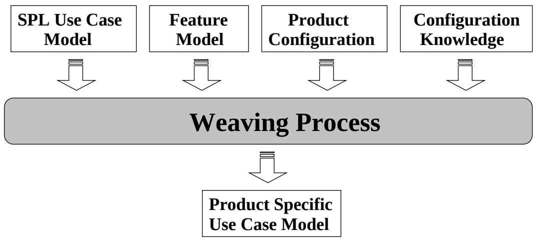


Figure 4: Overview of our weaving process.

3.1 Feature model

Feature models have an important contribution to our weaving process, since they are used for checking if a product configuration is a valid member of the product line. We have implemented specific functions in Haskell to deal with this kind of validation. Basically, these functions verify if all constraints defined in a feature model are satisfied by a product configuration.

For instance, below we present the signature of the *topmost* function for checking instances of a feature model. It states that `checkConfiguration` takes as input a feature model (FM) and a product configuration (PC), returning all constraints that a PC does not satisfy. If a PC complies with all constraints, the function returns an empty list.

$$checkConfiguration :: FM \rightarrow PC \rightarrow ErrorList$$

Going into details, the `checkConfiguration` function traverses two *trees of features*, representing both feature model and product configuration, at the same time that it calls specific functions for checking the rules applied for each type of feature relationship (mandatory, exclusive or, inclusive or) and constraints.

Moreover, auxiliary functions were defined for importing feature models, generated by the *Feature Modeling Plugin* (FMP) [4], to our environment. In the remainder of this section, we consider the *eShop* feature model introduced in the motivating example (Figure 1).

3.2 SPL use case model

This artifact defines scenarios that describe the expected behavior of the SPL's members. Scenarios might be optional, have parameters, and change (advice) the behavior of other scenarios. A use case model is composed by *use cases* and *aspectual use cases*. A use case has a name, a description and a list of scenarios, which are composed by a sequence of steps (pairs of *User action x System response*). An aspectual use case has a name and a list of advices, which can be used to extend the behavior of existing scenarios.

Regarding tool support, instances of the use case model can be written in any text editor that is able to export documents using a specific XML format[1]. Additional functions were developed for parsing these documents to the abstract data type of our use case model. In this running example, we consider the following scenarios and advices:

[1]Templates are available for Microsoft Word.

Proceed to Purchase: this mandatory scenario (Figure 5) specifies the common behavior that is required for confirming a purchase. Notice that a parameter (SM) is referenced in Step P2. This parameter allows the reuse of the *Proceed to Purchase* scenario for different configurations of the *Shipping Method* feature. The configuration knowledge artifact (Section 3.3) is responsible for relating parameters to features.

The annotation `[CustomerPreferences]` (Step P3) is another variation point of *Proceed to Purchase* scenario. This annotation, also assigned to the Step S3 of *Search for Products* (Figure 8), reveals points in the specification that are related to the customer preferences. Since pointcuts can make references to annotations, the advice *Register User Preferences* (Figure 9) extends the behavior of *Proceed to Purchase* after Step P3. Note that our annotations just reveal variation points, which means that annotated steps are *obliviousness* about which features extend the corresponding behavior.

Id: SC01
Description: `Proceed to purchase`

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 shipping methods (<SM>), 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. [CustomerPreferences]

Figure 5: Proceed to purchase scenario.

Buy Product: this advice (Figure 6) enables a customer to buy goods from an on-line shopping store. It is only available in the SPL members that are **not** configured with the *Shopping Cart* and *Bonus* features. This advice introduces an optional behavior **before** the join points identified in its *pointcut* clause. In this case, the Step P1 defined in the *Proceed to Purchase* scenario. Therefore, the *pointcut* clause can refer either to step ids or step annotations (as explained before). We encourage the definition of pointcuts based on annotations, since they mitigate the problem of fragile pointcuts. This problem occurs because changes in the order of steps might break pointcuts.

Id: ADV01
Description: `Buy a specific product`
Before: `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 6: Buy a product advice.

Buy Products with Shopping Cart and Bonus: this advice (Figure 7) allows purchasing products that have been previously added to a shopping cart. It extends the behavior of the *Proceed to Purchase* scenario by introducing the specific behavior required by the *Shopping Cart* and *Bonus* features. So, this advice is required by products that are configured with both *Shopping Cart* and *Bonus* features.

Id: ADV02
Description: `Buy products using a shopping-cart`
Before: `P1`

Id	User Action	System Response
C1	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.
C2	Select the confirm option.	Request bonus and payment information.

Figure 7: Buy products with shopping cart advice.

Search for Products: this mandatory scenario (Figure 8) allows a customer to search for products. In order to save space, we only present Step S3, which performs a search based on the input criteria. Note that, similarly to the Step P3 shown in Figure 5, this step is also assigned to the `[CustomerPreferences]` annotation. Therefore, any advice that points to this annotation extends, at least, the behavior of *Proceed to Purchase* and *Search for Product* scenarios.

Id: SC02
Description: `Search for products.`

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. [CustomerPreferences]

Figure 8: Search for products scenario.

Register User Preferences: this advice (Figure 9) updates the user preferences based on the user's history of searches and purchases. Its behavior can be started **after** any step assigned to the `[CustomerPreferences]` (see the *pointcut* clause) annotation and is available in products that are configured with the *Update User Preferences* feature.

Id: ADV03
Description: `Register user preferences.`
After: `[CustomerPreferences]`

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

Figure 9: Register user preferences.

At this point we are able to present some remarks related to our use case model. As discussed in Section 2, PLUSS and PLUC represent all valid configurations of a scenario in a single artifact. Differently, using our approach, we could separate the common behavior required to confirm a purchase (the base scenario *Proceed to Purchase*) from its variants, specified in the *Buy Product* and *Buy Product with Shopping Cart* advices.

Another distinction is that, in our approach, *feature* and *use case models* are syntactically independent from each other. There is no direct association joining these models and all relationships between them are kept by the configuration knowledge (Section 3.3). Actually, the weaving process, whose semantics are presented in Section 4, is responsible for binding the variation points defined in the SPL use case model. In order to do that, other assets (such as product configurations and configuration knowledge) have to be considered.

As a final remark, since different advices can make references to a same join point, interferences between advices may occur. Nevertheless, the weaving process evaluates each advice in a sequence defined by the configuration knowledge. For this reason, interferences between advices are deterministic and do not require specific constructs in the use case model to deal with them.

3.3 Configuration knowledge

This artifact corresponds to a list of configuration items, which relates feature expressions to tasks (or transformations) used for automatically generating a SPL member. Consequently, the configuration knowledge guides the weaving process. For this running example, the configuration knowledge shown in Table 1 enforces that

- Both *Proceed to Purchase* (SC01) and *Search for Products* (SC02) scenarios are mandatory, since their selection is related to the (mandatory) root feature of the *eShop* product line;
- The *Buy Product* advice (ADV01) is used in the composition of products that do not have been configured with both *Shopping Cart* and *Bonus* features— if both features were selected for a product, it would be configured with the *Buy Product with Cart* advice (ADV02). Remember that there is a mutual dependency between *Shopping Cart* and *Bonus* features. As a consequence, no product can be configured with just one of them;
- The *Register User Preferences* advice (ADV03) is not used in a product composition unless the *Update User Preferences* feature has been selected; and
- References to the "SM" parameter are bound to the selected alternatives of the *Shipping Method* feature.

Three different tasks are shown in Table 1: *select scenario*, *evaluate advice*, and *bind parameter*. According to Bachmann et al. [6], the first type of task deals with the source of variability named as *variation in function*, being responsible for selecting scenarios; the second one deals with *variation in control flow*, being responsible for composing advices at specific join points; and the last one deals with *variation in data*, being responsible for binding parameters to features.

Feature Expression	Tasks
eShop	select scenario **SC01** select scenario **SC02**
not (Shopping Cart **and** Bonus)	evaluate advice **ADV01**
(Shopping Cart **and** Bonus)	evaluate advice **ADV02**
Update User Preferences	evaluate advice **ADV03**
Shipping Method	bind **SM** to **Shipping Method**

Table 1: Example of configuration knowledge

The semantics of these tasks are described as crosscutting mechanisms in Section 4. Besides that, the configuration knowledge data type is shown bellow. As explained before, it is a list of configuration items, which relate feature expressions to *tasks*. Actually, the type `Task` is a synonymous for the family of functions that receive a *SPL use case model* (SPL) and a *product specific use case model* ($SPLMember$) as parameters; and then returns a refinement of the product specific model.

```
type ConfigurationKnowledge = [Configuration]
data Configuration = Configuration {
    exp :: FeatureExpression,
    tasks :: [Task]
}
type Task = (SPL → SPLMember) → SPLMember
```

Note that feature expressions have to be written in propositional logic, because it is necessary to express, for example, that the *Buy Products* advice will be evaluated iff the feature expression **"not ShoppingCart and Bonus"** is true for a specific product configuration.

Indeed, the *weavingProcess* function, whose source code is presented bellow, behaves like a generator that applies the appropriate list of tasks (*ts*) for a given product configuration. This means that, first of all, it is necessary to verify (*eval pc* (*exp c*)) which expressions (*exp c*) are valid for a specific product configuration (*pc*). After that, the $stepRefinement$ function composes the sequence of tasks that must be applied.

```
weavingProcess spl fm pc ck =
    stepRefinement [(t spl) | t ← ts] empty
    where
        ts = concat [tasks c | c ← ck, eval pc (exp c)]
        empty = (emptyInstance spl pc)
        stepRefinement l m = ...
```

As a consequence, supposing the list of tasks

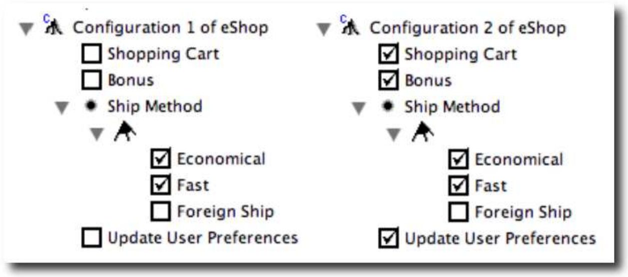
$ts = [select(SC01), evaluate(ADV01), evaluate(ADV02)]$,

the semantics of this hypothetical product would be given by: $p = f\ (spl,\ g\ (spl,\ h))$, where:

$$\begin{aligned} h &= (select\ SC01)\ spl\ emptyInstance \\ g &= (evaluate\ ADV01) \\ f &= (evaluate\ ADV02) \end{aligned}$$

3.4 Product configuration

This artifact identifies a specific SPL member, which is characterized by a configuration of features. One important property is that each product configuration must comply to the relationships and constraints specified in the corresponding feature model. Product configurations may be also represented using the *Feature Modeling Plugin* [4]. Auxiliary functions parse such configurations to our environment, instantiating the abstract representation of the *product configuration*— basically a tree representing a selection of features. For the *eShop* example, two possible configurations are shown in Figure 10.


Configuration 1 of eShop
- [ ] Shopping Cart
- [ ] Bonus
- ● Ship Method
  - [✓] Economical
  - [✓] Fast
  - [ ] Foreign Ship
- [ ] Update User Preferences

Configuration 2 of eShop
- [✓] Shopping Cart
- [✓] Bonus
- ● Ship Method
  - [✓] Economical
  - [✓] Fast
  - [ ] Foreign Ship
- [✓] Update User Preferences

Figure 10: Examples of product configurations.

The first configuration (on the left side of the Figure 10) defines a product that does not have support for *Shopping Cart*, *Bonus* and *Update User Preferences*. Additionally, it supports only the economical and fast shipping methods. The second configuration is more complete, being configured with the features *Shopping Cart*, *Bonus*, and *Update User Preferences*.

As explained in the previous section, the features selected to a specific product (represented as a product configuration) identify which *tasks* (or transformations) must be performed in order to generate the corresponding SPL member. Figure 11 depicts the resulting scenarios after evaluating the configuration knowledge of Table 1 and considering the second configuration of Figure 10. According to what was explained in the previous section, such a configuration requires:

1. The selection of *Proceed to Purchase* and *Search for Product* scenarios. This selection is directly responsible for the resulting scenarios shown in Figure 11.
2. The evaluation of *Buy Products with Cart* advice. The result of this evaluation is the introduction of the first two steps in the resulting specification of the *Proceed to Purchase* scenario.
3. The evaluation of *Register User Preferences* advice. The result of this evaluation is the introduction of the last steps on both *Search for Products* and *Proceed to Purchase* scenarios.
4. The binding of the *SM* parameter to the selected options of the *Shipping Method* feature. This binding is responsible for setting the options *Economical* and *Fast* in the fourth step of the resulting *Proceed to Purchase* scenario.

Scenario: `Search for products.`

id	User Action	System Response
...	...	...
3	Inform the search criteria.	Retrieve the products that satisfy the search criteria. Show a list with the resulting products.
4	-	Update the preferences based on the search results or purchased items

Scenario: `Proceed to purchase.`

id	User Action	System Response
1	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.
2	Select the confirm option.	Request bonus and payment information.
3	Fill in the requested information and select the proceed option.	Request the shipping method and address.
4	Select one of the available shipping methods (**Economical, Fast**), fill in the destination address and proceed.	Calculate the shipping costs.
5	Confirm the purchase.	Execute the order and send a request to the Delivery System to dispatch the products.
6	-	Update the preferences based on the search results or purchased items.

Figure 11: Resulting scenarios of the example

4. MODELING FRAMEWORK

In this Section we describe the notation proposed to represent variability management as crosscutting mechanisms. In fact, our notation is a slight customization of the *Crosscutting Modeling Framework*, proposed by Masuhara and Kiczales (the MK framework) [20].

The goal of the MK framework is to explain how different *aspect-oriented* mechanisms support crosscutting modularity. In order to do that, each mechanism is represented 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 [20]. Their requirement for characterizing a mechanism as crosscutting is fulfilled by our approach, in the sense that different specifications contribute to the definition of a SPL member.

Similarly to the MK work, our modeling framework represents each weaver as an 6-tuple (Eq. 1 and Table 2), highlighting the contribution of each input language in the composition processes. We represent each weaver by filling in the six parameters of our 6-tuple representation, by providing a reference implementation for each weaver, and by stating how elements of the weaver implementation correspond to elements of the model.

$$W = \{o, o_{jp}, L, L_{id}, L_{eff}, L_{mod}\}, \tag{1}$$

Table 2: 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

Next, we describe the semantics of our weaving process, modeling one weaver for each source of variability and providing reference implementations for them.

4.1 Variability in function

Variability in function occurs when a particular function might exist in some products and not in others [6]. For this source of variability, a corresponding weaver is responsible for selecting scenarios based on specific product configurations.

This weaver is implemented by the function *selectScenarios* (next code fragment). It takes as input the list of *scenario ids* that should be selected for a specific feature expression, the *SPL use case model* (spl), and the *product* being generated. Then, this function returns a new configuration of the product, which is refined by each scenario s in the SPL use case model that satisfies the condition $(id\ s) \in\ ids$. In this case, the configuration knowledge (CK) and the SPL use case model (UCM) crosscut each other with respect to the list of selected scenarios of a specific SPL member.

```
$selectScenarios\ ids\ spl\ product =$
$addScenarios\ (product, scenarios)$
**where**
  $scenarios = [s \mid s \leftarrow (splScenarios\ spl), (id\ s)\ \in\ ids]$
  $addScenarios\ ...$
```

Below we present the concrete instantiation of the first line of the *eShop* configuration knowledge (Table 1), which deals with this source of variability. Notice that the configuration knowledge binds the first parameter of the *selectScenario* function. Therefore, particularly to this weaver, the configuration knowledge contributes to the identification of the scenarios that must be selected to a specific product.

```
$c1 = Configuration\ (``eShop",$
  $selectScenarios\ [``SC01", ``SC02"])$
```

The model of the *Variability in Function Weaver*, in terms of the framework, is shown in Table 3. The *selectScenarios* function is used to argue that the model is realizable and appropriate [20]. We achieve this by matching the model elements to the corresponding parameters and auxiliary functions in the implementation.

The input languages *configuration knowledge* (CK) and *SPL use case model* (UCM) contributes to the binding of *selectScenarios* parameters. An instance of the SPL use case model corresponds to the specification of all SPL scenarios. Instead, the identification of which scenarios must be selected to a specific feature expression are documented in the configuration knowledge. Finally, the *product configuration* (PC) specifies which features were selected for a specific product.

Table 3: Model of Product Derivation

Element	Description
o	Product specific scenarios (list of scenarios)
o_{jp}	Scenario declarations
L	{UCM, CK, PC}
UCM_{ID}	SPL scenarios
CK_{ID}	Feature expressions and scenario IDs
PC_{ID}	Product specific feature selection
UCM_{EFF}	Provides declaration of scenarios
CK_{EFF}	Relates feature expressions to scenario Ids
PC_{EFF}	Triggers scenario selection
UCM_{MOD}	Scenario
CK_{MOD}	Each configuration item
PC_{MOD}	Each selected feature

4.2 Variability in data

This kind of variability occurs whenever two or more scenarios share the same behavior (the sequence of steps) and differ in relation only to values of a same concept. For instance, the *Proceed to Purchase* scenario (Figure 5) can be reused for different kinds of shipping method.

The function *bindParameter* is the reference implementation of this weaver. It receives as input a parameter identifier (pId); a feature identifier (fId); the SPL use case model (*spl*); and the *product* being generated. Then, it replaces all references to pId in *product* by a suitable representation of the corresponding feature selection ($features$). For example, if a product is configured with both *Economical* and *Fast* shipping methods, applying this weaver for the *Proceed to Purchase* scenario (Figure 5) replaces each reference to the "SM" parameter by the text (*Economical, Fast*).

```
bindParameter pId fId spl product =
  bindParameter' product steps options
  where
    features = (configuration product)
    steps = [s | s ← (steps spl), (s 'refers' pId)]
    options = selectedOptions (features, fId)
    bindParameter' ...
```

Below we present the concrete instantiation of the fifth line of the *eShop* configuration knowledge (Table 1), which deals with this source of variability. In this case, the configuration knowledge binds the first two parameters of the *bindParameter* function, contributing as a mapping between *parameter ids* and feature configurations. Such a mapping reduces the coupling between SPL use case models and feature models.

```
c5 = Configuration ("ShippingMethod",
  bindParameter ("SM", "ShippingMethod"))
```

Table 4 represents the model of *Variability in Data*. Since this weaver solves parameters in scenario specifications, its output language is a list of product specific scenarios.

Table 4: Model of Bind Parameters Weaver

Element	Description
o	Scenarios with resolved parameters
o_{jp}	Each resolved parameter
L	{UCM, CK, PC}
UCM_{ID}	Parameterized steps
CK_{ID}	Mapping of parameters to features
PC_{ID}	Selected features related to a parameter
UCM_{EFF}	Declares parameterized scenarios
CK_{EFF}	Relates parameters to features
PC_{EFF}	Defines the domain value of parameters
UCM_{MOD}	Use case scenarios
CK_{MOD}	Each configuration item
PC_{MOD}	Each selected feature

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 product configuration (PC), in the sense that the domain values of a parameter is defined (PC_{EFF}) in the product specific features; and from the configuration knowledge (CK), which is used for relating parameters to features (CK_{EFF}).

4.3 Variability in control flow

This source of variability occurs when a particular pattern of interaction (a use case scenario) varies from one product to another. Differently from PLUSS and PLUC, in our approach we use the notion of *advices* to modularize the variant behavior of a scenario. An advice customizes the specification of a feature, by means of extending the common behavior of existing scenarios with additional steps. The flow of events of an advice can be inserted before or after the steps referenced by its pointcut clause, which can be either the *step id* or *annotations* assigned to different steps.

The function *evaluateAdvice* is the reference implementation of this weaver. Basically, this mechanism takes as input the name of an advice to be evaluated; the SPL use case model (*spl*), which defines both use cases and advices; and the *product* being generated. Then, it evaluates the advice, retrieving the steps in *product* that match the pointcut clause. Finally, the function introduces the advice's flow of events before or after the matched steps.

```
evaluateAdvice name spl product =
  evaluateAdvice' product advice
  where
  advice = head [a | a ← advices spl, (id a) == name]
evaluateAdvice' product advice =
  if (isBefore advice)
    then composeBefore product matchedSteps flow
    else composeAfter product matchedSteps flow
  where
    matchedSteps = match (product (pointCut advice))
    flow = scenarioOfAdvice (advice)
```

Below we show a concrete instantiation of the second line of the *eShop* configuration knowledge (Table 1). Notice that the configuration knowledge binds the first parameter of the *evaluateAdvice* function, being responsible for relating features expressions to advices.

$$c3 = Configuration\ (\text{“}ShoppingCart''\ \texttt{'And'}\ \text{“}Bonus'',\ evaluateAdvice\ (\text{“}ADV02''))$$

The model of this weaver is in Table 5. The output is a new configuration of the *product specific use case model*, whose scenarios were extended by alternative (or optional) flows of events. The extensions occur before or after the steps that match (O_{JP}) the pointcut clause (UCM_{ID}) of the advice, declared in the SPL use case model (UCM). The effect of evaluating advices in the composition process is to extend product specific scenarios that may not define a concrete flow of events.

Table 5: Model of Scenario Composition Weaver

Element	**Description**
o	Specific scenarios with extensions
o_{jp}	Scenarios and steps of scenarios
L	{UCM, CK, PC}
UCM_{ID}	Pointcuts of declared advices
CK_{ID}	Mapping of features to advices
PC_{ID}	Selected features related to advices
UCM_{EFF}	Provides declaration of advices
CK_{EFF}	Relates features to advices
PC_{EFF}	Triggers advice selection
UCM_{MOD}	Advices
CK_{MOD}	Each configuration item
PC_{MOD}	Each selected feature

5. EVALUATION

We have applied our approach to four SPLs: the eShop Product Line, as partially shown in Section 3; the Pedagogical Product Line (PPL) [1], which was proposed for learning about and experimenting with software product lines, and focuses on the arcade game domain; the Smart Home Product Line, based on different specifications [23, 3], and the Multimedia Message Product Line (MMS), a case study conducted with one of our industrial partners. In this paper we focused our comparisons with the PLUSS approach, mainly because in a previous work we have identified that PLUC specifications are not maintainable at all [8].

It is important to note that each case study was conducted with different settings. For instance, groups of students were assigned to specify the behavior of the Smart Home product line in both PLUSS and MSVCM techniques. Differently, we compared our approach to an available PLUSS specification of the PPL. The input data and the tasks performed in each case study were also different. For example, some of the case studies (Smart Home and MMS) started from the specification of different products of a family. On the other hand, both *eShop* and PPL case studies started from existing product line specifications. Next, we first present this new metric suite, and then describe the assessment of the Smart Home, MMS, and PPL case studies.

5.1 Metric suite

In order to evaluate our approach, we customized the metric suite proposed by Eaddy et al. [12], considering the degree of scattering of features and the degree of focus of scenarios. We also customized their *prune dependency analysis*, as a guide to assign features to scenario steps. Actually, we consider that a step s depends on a feature f iff the configuration of f effects the selection or the configuration of the step s. Therefore, we name our assignment approach *configuration dependency analysis*. The *degree of scattering* of a feature f is calculated by normalizing its concentration with respect to each scenario $s \in S$ (the set of all scenarios).

$$DOS(f) = 1 - \frac{|S|\sum_{s}^{S}(CONC(f,s) - \frac{1}{|S|})^2}{|S|-1}, \text{ where:}$$

$$CONC(f,s) = \frac{number\ of\ steps\ in\ s\ assigned\ to\ f}{total\ number\ of\ steps\ assigned\ to\ f}$$

Likewise, the *degree of focus* of a scenario s is calculated by normalizing its dedication with respect to each feature $f \in F$ (the set of all features).

$$DOF(s) = \frac{|F|\sum_{f}^{F}(DEDI(s,f) - \frac{1}{|F|})^2}{|F|-1}, \text{ where:}$$

$$DEDI(s,f) = \frac{number\ of\ steps\ in\ s\ assigned\ to\ f}{total\ number\ of\ steps\ in\ s}$$

These metrics inherit the same properties of the original ones [12]: (a) the *degree of scattering* (DoS) is normalized between 0 (completely localized) and 1 (completely unlocalized); and (b) the *degree of focus* (DoF) is also normalized between 0 (completely unfocused) and 1 (completely focused).

For instance, consider the scenarios shown in Figure 12, which represents the assignment of a few Smart Home features to both PLUSS and MSVCM specifications. The left hand side of Figure 12(a) depicts that the PLUSS specification of Register Inhabitant scenario has seven steps. Two of these steps were assigned only to the *Register Inhabitant* (RI) feature; other two steps were assigned to the interactions between *Register Inhabitant* and *Password* features; and other three steps were assigned to the interactions between *Register Inhabitant* and *Fingerprint* features. Notice that the *Password* and *Fingerprint* features, which are mutually exclusive, also change the behavior of the *Request Access to Home* feature (Figure 12(b)).

Similarly, Figure 13 shows some feature assignments for the MMS product line. In this case, we want to emphasize that the behavior related to the *Store an Embedded Number* and *Send a Message to Embedded Email* features change the behavior of the *Receive a MMS* and *Select a MMS for Displaying* features in a similar way. For this reason, we classify the behavior of the *Store an Embedded Number* and *Send a Message to Embedded Email* features as *homogeneous*. On the other hand, we say that the *Password* optional feature of the Smart Home case study has a *heterogeneous crosscutting behavior* [5], since it requires one advice for each join point (Figure 12(a) and (b)). The same occurred with the *Fingerprint* feature.

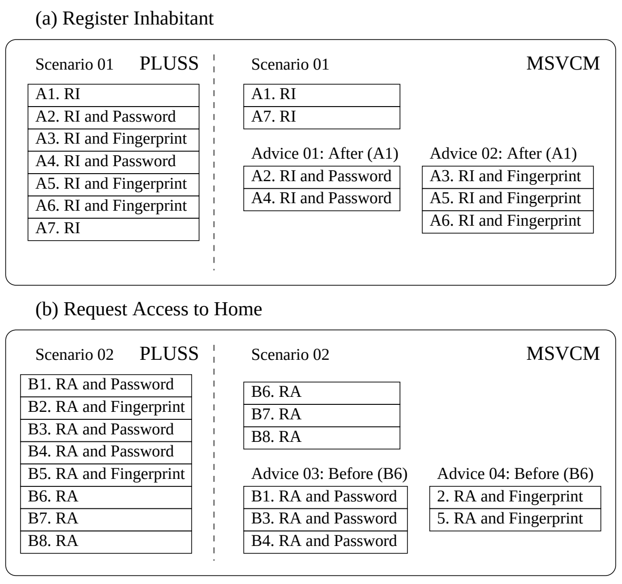


Figure 12: Sample of Smart Home assignments.

We have observed that the *crosscutting nature* of a feature, which here we consider as being homogeneous or heterogeneous [5], has a great influence in our results. For instance, Table 6 shows the DoS of the assets depicted in Figures 12 and 13. Notice that, in MSVCM, modularizing the behavior of *Password* and *Fingerprint* as advices had increased the DoS of both *Register Inhabitant* and *Request Access to Home* features. A similar problem does not occur with homogeneous crosscutting features— modularizing the *Store an Embedded Number* and *Send Message to Embedded Email* as advices does not change the DoS of the *Receive a MMS* and *Select a MMS for Displaying* features.

Table 6: DoS of the presented features

SPL	Feature	PLUSS	MSVCM
Smart Home	Register Inhabitant	0.00	0.57
	Request Access to Home	0.00	0.57
	Password	0.96	0.78
	Fingerprint	0.96	0.78
MMS	Receive a MMS	0.00	0.00
	Select a MSS for Displaying	0.00	0.00
	Store an Embedded Number	1.00	0.00
	Send a Message to Email	1.00	0.00

Besides that, independently of the crosscutting nature of a feature, the MSVCM approach eliminates the tangling of the base scenarios (Table 7), quantified by the DoF metric. In what follows, we present summaries of the data collected from the different case studies. All specifications and collected data are available [2].

Table 7: DoF of the presented scenarios

SPL	Scenario	PLUSS	MSVCM
Smart Home	Scenario 01	0.24	1.00
	Scenario 02	0.27	1.00
MMS	Scenario 01	0.12	1.00
	Scenario 02	0.20	1.00

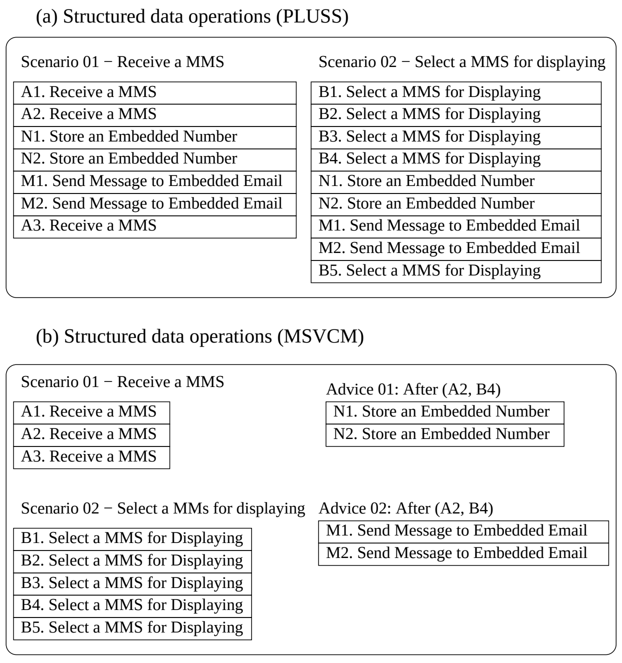

(a) Structured data operations (PLUSS)
Scenario 01 − Receive a MMS
A1. Receive a MMS
A2. Receive a MMS
N1. Store an Embedded Number
N2. Store an Embedded Number
M1. Send Message to Embedded Email
M2. Send Message to Embedded Email
A3. Receive a MMS
Scenario 02 − Select a MMS for displaying
B1. Select a MMS for Displaying
B2. Select a MMS for Displaying
B3. Select a MMS for Displaying
B4. Select a MMS for Displaying
N1. Store an Embedded Number
N2. Store an Embedded Number
M1. Send Message to Embedded Email
M2. Send Message to Embedded Email
B5. Select a MMS for Displaying
(b) Structured data operations (MSVCM)
Scenario 01 − Receive a MMS
A1. Receive a MMS
A2. Receive a MMS
A3. Receive a MMS
Advice 01: After (A2, B4)
N1. Store an Embedded Number
N2. Store an Embedded Number
Scenario 02 − Select a MMs for displaying
B1. Select a MMS for Displaying
B2. Select a MMS for Displaying
B3. Select a MMS for Displaying
B4. Select a MMS for Displaying
B5. Select a MMS for Displaying
Advice 02: After (A2, B4)
M1. Send Message to Embedded Email
M2. Send Message to Embedded Email

Figure 13: Sample of MMS assignments.

5.2 Smart Home assessment

This study aimed at comparing feature modularity of PLUSS and MSVCM specifications. Initially, three different products of the security module of a Smart Home family [23] were specified. Almost six use cases and fifteen scenarios are present in each product, with a significant number of duplicated steps among them. These specifications, available on-line, were used as input data. Then, two groups of students were assigned to restructure the input specifications, using either PLUSS or MSVCM approaches.

Table 8 summarizes the resulting *degree of scattering* of the Smart Home features, computed from the PLUSS and MSVCM specifications. Although the MSVCM approach achieved a central tendency (median) closer to zero, for some features (3 features in a total of 17) the DoS for the MSVCM approach presented a greater value than the corresponding ones in PLUSS.

Table 8: Smart Home degree of scattering

	Min	Median	Mean	Max	St. Deviation
PLUSS	0.00	0.26	0.28	0.70	0.08
MSVCM	0.00	0.00	0.26	0.82	0.10

As discussed in the previous section, modularizing the variant behavior of *Register Inhabitant* and *Request Access to Home* in advices produced the side effect of increasing the DoS, at the same time that it reduced the tangling of the related scenarios. On the other hand, the feature *Turn On Internal and External Lights* presents a lower DoS in the MSVCM approach (0.00) when compared to the PLUSS (0.54) notation. This feature requires a behavior that crosscuts, in a homogeneous way, the different scenarios related to the *Intrusion Detection* use case.

Considering the degree of focus (data summarized in Table 9), we have evidences of a significant improvement of the MSVCM approach (*p-value=0.06*). Notice that the central tendency (median) of DoF in the MSVCM approach (1.00) is closer to one than the corresponding value (0.63) in the PLUSS notation.

Table 9: Smart Home degree of focus

	Min	Median	Mean	Max	St. Deviation
PLUSS	0.33	0.63	0.68	1.00	0.07
MSVCM	0.35	1.00	0.82	1.00	0.05

5.3 MMS assessment

As explained earlier, the MMS product line, which was adapted from an industrial partner, 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) [8]. We have specified the MMS product line using both Crosscutting and PLUSS techniques. The results of this case study are summarized in Tables 10 and 11.

Table 10: MMS degree of scattering

	Min	Median	Mean	Max	St. Deviation
PLUSS	0.00	0.60	0.46	0.80	0.11
MSVCM	0.00	0.41	0.34	0.79	0.10

This case study, differently from the Smart Home, gives evidence of improvements of the MSVCM approach with respect to the feature's *degree of scattering* (*p-value=0.01*). This result was mainly achieved because several of the features of the MMS product line require a homogeneous crosscutting behavior. For example, there are some policies related to DRM[2] content that crosscut, in a uniform manner, the behavior related to *sending* and *forwarding* messages.

On the other hand, the MMS case study didn't reveal significant improvements with respect to the *degree of focus* metric (Table 11). The main reason for such a result is that just a few scenarios of the MMS case study are effected by alternative features. Thus, the benefits achieved from extracting varying behavior to aspectual scenarios were minimized in this case study.

Table 11: MMS degree of focus

	Min	Median	Mean	Max	St. Deviation
PLUSS	0.34	0.59	0.70	1.00	0.07
MSVCM	0.46	0.59	0.71	1.00	0.06

5.4 PPL assessment

We compared our approach to a PPL specification that was sent to us by the authors of the PLUSS technique. Similarly to the Smart Home product line, the PPL has been used in several case studies in the area. The original specification of PPL [1] is already well modularized, since its features, in general, do not crosscut different use cases. Moreover, another characteristic of the PPL is that several features are related to qualities that do not effect scenario specifications.

[2]Digital Rights Management

Even in this context, our approach achieves some improvements in the resulting *degree of scattering* (Table 12). Modularizing the error handling feature was the main reason for the aforementioned improvement. By applying our approach, all behavior related to the *error handling* concern was modularized in a single advice.

Table 12: PPL degree of scattering

	Min	Median	Mean	Max	St. Deviation
PLUSS	0.00	0.48	0.32	0.71	0.08
MSVCM	0.00	0.00	0.07	0.64	0.05

Similarly to the MMS case study, we didn't get significant improvements in the *degree of focus* metric (Table 13). Again, this result was primarily motivated by the reason that just a few scenarios of PPL are effected by alternative features.

Table 13: PPL degree of focus

	Min	Median	Mean	Max	St. Deviation
PLUSS	0.44	1.00	0.78	1.00	0.07
MSVCM	0.44	1.00	0.83	1.00	0.07

5.5 Discussion

After running the different case studies, we concluded that the *greater* is the number of *homogeneous crosscutting features* and the number of variants for a scenario, the greater is the benefits of applying our approach. In that cases, the MSVCM improvements on both feature modularity and scenario cohesion become more evident.

In fact, the expected benefit of our approach is to reduce the impact of evolving a SPL. The metric suite presented here tries to quantify such an impact, since (a) if a scenario has a lower DoF, introducing a new alternative to its variant behavior requires more changes in the base specification; and (b) evolving features with high DoS requires changes in different scenarios. Although the significance level varies from one case study to another, in most of the cases we could improve both scenario cohesion and feature scattering. Therefore, the modularity supported by our approach reduces the number of changes in the base scenarios of a SPL; in such a way that it can evolve by means of the introduction of new scenarios and/or advices.

It is important to note that SPL development, independently of the approach, requires a mapping between the domain and solution spaces [11, 18]. As a consequence, in this paper we did not introduce new kinds of concerns to the SPL development. Actually, our approach mainly presents a better separation for mapping those SPL concerns.

6. RELATED WORK

Several approaches have been proposed for representing scenario variability [17, 13, 7]. However, in this paper we focused on PLUC and PLUSS techniques because they encompass a broad range of SoC between variability management and scenario specifications. PLUC presents the lowest level of modularity, since almost all information related to variability is tangled within use cases. Although PLUSS partially reduces such a tangling, by considering the importance of feature modeling, some dependencies from scenarios to features are still present.

There are works proposed to represent weaving mechanisms for textual requirements [9, 25]. However, these approaches offer a narrow support to the *control flow* source of variability, requiring enhancements to integrate their weaving mechanisms to variability management. Besides, generic approaches for *aspect-oriented modeling* (AOM) have been applied for variability management [18, 22, 16]. For instance, Jayaraman and others proposed an approach for modeling variability in UML diagrams [18]. Based on their approach, SPL members are generated by means of *graph transformations* specified in MATA (Modeling Aspects using a Transformation Approach). Differently, we proposed a framework for modeling variability as crosscutting mechanisms; and instantiated such a framework for designing variability mechanisms for *textual scenarios*. Moreover, instead of graph transformations, we use an interpreter based approach for SPL member generation— actually, each reference implementation provided for the weaving process is an interpreter.

Finally, we could have used absolute values for quantifying scattering and tangling, as in a previous work [8]. However, absolute values, such as proposed by Figueiredo and others [14], just reveal if a feature is scattered or not— without any information about the degree of its scattering. In fact, this limitation hinders the comparison of modularity between different specifications. As a consequence, we adopted a suite of metrics for quantifying *degree of scattering* and *degree of focus* [12].

7. CONCLUSIONS

In this paper we formally described variability management as crosscutting mechanisms, 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 a clear separation of concerns between variability and scenario specifications and allowing both representations to evolve independently. Moreover, for different case studies our approach improved both feature modularity and scenario cohesion.

8. ACKNOWLEDGMENTS

We gratefully acknowledge Mehmet Aksit and the anonymous referees for useful suggestions to improve the paper. This research was partially sponsored by CNPq (grant CT-INFO 17/2007).

9. REFERENCES

[1] Arcade game maker pedagogical product line. http://www.sei.cmu.edu/productlines/ppl/.

[2] Software productivity group. online: http://www.cin.ufpe.br/spg.

[3] M. Alferez et al. A model-driven approach for software product lines requirements engineering. In *SEKE' 2008*, pages 779–784, San Francisco, USA, 2008.

[4] M. Antkiewicz and K. Czarnecki. Feature plugin: feature modeling plug-in for eclipse. In *OOPSLA workshop on eclipse technology eXchange*, pages 67–72, New York, NY, USA, 2004.

[5] S. Apel, T. Leich, and G. Saake. Aspectual mixin layers: aspects and features in concert. In *ICSE' 2006*, pages 122–131, New York, NY, USA, 2006.

[6] F. Bachmann and L. Bass. Managing variability in software architectures. *SIGSOFT Softw. Eng. Notes*, 26(3):126–132, 2001.

[7] A. Bertolino and S. Gnesi. Use case-based testing of product lines. In *ESEC/FSE' 2003*, pages 355–358, Helsinki, Finland, 2003.

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

[9] R. Chitchyan, A. Rashid, P. Rayson, and R. Waters. Semantics-based composition for aspect-oriented requirements engineering. In *AOSD' 2007*, pages 36–48, New York, NY, USA, 2007.

[10] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[11] K. Czarnecki and U. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co, 2000.

[12] M. Eaddy, A. Aho, and G. Murphy. Identifying, assigning, and quantifying crosscutting concerns. In *First Workshop on Assessment of Contemporary Modularization Techniques (ACOM)*, Minneapolis, USA, May 2007.

[13] M. Eriksson, J. Borstler, and K. Borg. The PLUSS approach - domain modeling with features, use cases and use case realizations. In *SPLC' 2005*, pages 33–44, Rennes, France, 2005.

[14] E. Figueiredo et al. Evolving software product lines with aspects: an empirical study on design stability. In *ICSE' 2008*, pages 261–270, Leipzig, Germany, 2008.

[15] 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.

[16] I. Groher and M. Voelter. Using aspects to model product line variability. In *Early Aspects Workshop at SPLC*, 2008.

[17] I. Jacobson, M. Griss, and P. Jonsson. *Software reuse: architecture, process and organization for business success*. Addison-Wesley Publishing Co., 1997.

[18] P. Jayaraman et al. Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis. *LNCS*, 4735:151–165, 2007.

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

[20] H. Masuhara and G. Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In *ECOOP' 2003*, pages 2–28, 2003.

[21] A. M. D. Moreira and J. Araújo. Handling unanticipated requirements change with aspects. In *SEKE' 2004*, pages 411–415, Alberta, Canada, 2004.

[22] B. Morin et al. A generic weaver for supporting product lines. In *Early Aspects Workshop at ICSE*, pages 11–18, Leipzig, Germany, 2008.

[23] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering : Foundations, Principles and Techniques*. Springer, 2005.

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

[25] J. Sillito et al. Use case level pointcuts. In *ECOOP' 2004*, pages 244–266, Oslo, Norway, 2004.