Scenario Variability as Crosscutting

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 describing or implementing different kinds of variability that might occur at different SPL views (requirements, design, implementation, and test). In this paper, we present an 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 separated way. We achieve such goal by modeling variability management as a crosscutting phenomenon, for the reason that artifacts such as feature models, product configurations, and configuration knowledge crosscut each other with respect to a SPL specific member. Additionally, we 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.

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, requirement models

1. INTRODUCTION

The support for variation points enables product customization from a set of reusable assets [30]. However, variability management, due to its inherent crosscutting nature, is a common challenge in software product line (SPL) adoption [11, 30]. 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 is often spread not only in several artifacts, but also at the different product line views (requirements, design, implementation, and test). So, 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 [30].

Several authors have proposed the use of *aspect-oriented* mechanisms in order to better modularize the composition of common and variant assets of a product line [32, 12, 16, 4, 3, 5]. However, here we go beyond this composition issue. We mainly consider a more encompassing notion of variability management, including artifacts such as feature models [19, 14, 25] and configuration knowledge [14, 30], and explain it as a crosscutting phenomenon.

In this paper, we apply this view of *variability management as crosscutting* for modularizing SPL use case scenarios. Current techniques for scenario variability management [21, 7, 16] do not present a clear separation between variability management and scenario specification. For instance, supposing that details about product variants are tangled with use case scenarios, if one variant is removed from the product line, it is necessary to change all related scenarios. In summary, it is difficult to evolve both representations. Another problem is the lack of a formal representation for the derivation of product specific scenarios. This is not suitable for current SPL generative practices [26]; following the aforementioned works, it is difficult to automatically derive the requirements of a specific SPL member and to check if the specified compositions between common and variant flows are correct. Although the semantic composition of PLUC (Product Line Use Case) [7] is defined in [17], such approach does not separate scenario specification from variability management, as presented in Section 2.

In order to formalize our approach, we describe a framework for modeling the composition process of scenario variabilities with feature models, product configuration, and configuration knowledge (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. Therefore, the main contributions of this work are

- Characterize the broader notation of variability management as a crosscutting concern and, in this way, propose an approach where it is 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. Actually, it has already been claimed for source code [3, 5], without considering the importance of variability artifacts.
- A framework for modeling the composition process of scenario variability mechanisms. This framework gives a basis for describing variability mechanisms (such as parameterization), allowing a better understanding of each mechanism. In this work, such framework is used for modeling the semantics of scenario variability mechanisms, but it might be customized for other SPL views.
- Describe three scenario variability mechanisms (selection of optional scenarios, scenario composition, scenario parameterization) using the modeling framework. Such descriptions present a more formal representation when compared to existing works; this is an important property for supporting the automatic derivation of product specific artifacts.

Since our concept of crosscutting mechanism is based on Masuhara and Kiczales work [27], another contribution of this paper is that we apply their ideas to the languages of variability management, reinforcing the generality of their model, which was originally instantiated only for mechanisms of aspect-oriented programming languages. Based on their view of crosscutting, we can reason about variability management as a crosscutting concern that involves different input specifications that contribute to derive a specific member of a given SPL.

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

2. MOTIVATING EXAMPLE

In order to customize specific products, by selecting a valid feature configuration, variantion points must be represented in the product line artifacts. Several variant notations have been proposed for use case scenarios, like Product Line Use Cases (PLUC) [7] and Product Line Use Case Modeling for Systems and Software Engineering (PLUSS) [16]. 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* [31] 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 [31]. Here, we consider extra features, such as *Update User Preferences*, which, based on the user historical data of searches and purchases, updates user preferences. Figure 1 presents part of the *eShop* feature model [19, 14, 25]; the relationships between a parent and its child 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), *Or* (one or more subfeatures might be selected), and *Alternative* (exactly one subfeature must be selected for each product).

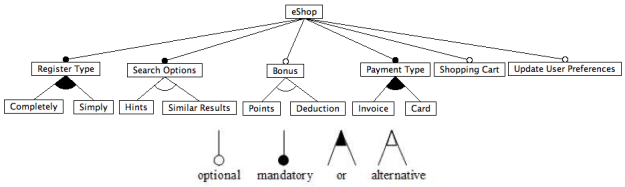

Figure 1: eShop feature model.

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 2 are never performed together. They are alternative steps: Step 1(a) will be present only if the *Shopping Cart* feature is selected (otherwise Step 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. In this technique, such restrictions are documented in feature models.

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

Figure 2: Buy Products scenarios using the PLUSS notation.

Following this approach, it is hard to understand the behavior of a specific product 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 the *Buy Product* scenario, enabling the search for product availability in remote warehouses (a new variant for Step 1) and updating a remote warehouse when the user confirms the purchase (a new variant for Step 5). Moreover, the inclusion of this new optional feature also changes the specification of the *Search for Products* scenario (the search might also be remote). In summary, since the behavior of certain features may be spread among several specifications and each specification might describe several variants, the effort needed to understand and evolve the product line might increase.

Instead of relating each variant step to a feature, PLUC introduces special tags for representing variabilities in use case scenarios. For example, the VP1 tag in Figure 3, which also describes the *Buy Products* scenario, denotes a variation point that might assume the values "*checkout*" or "*buy item*", depending on which product is configured. For each *alternative* or *optional* step, one tag must be defined. The actual value of each tag is specified in the *Variation Points* section of the scenario specification.

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

    Products definition:
    VP0 = (P1, P2)

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

Figure 3: Buy Products scenarios using the PLUC notation.

A different 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, however, is not configured with such features.

Since the values of alternative and optional variation points are computed based on the defined products, instead of specific features, the inclusion of a new member in the product line might require a deep review of variation points. Moreover, since the variation points and the product definitions are spread among several scenario specifications, it is hard and time consuming to keep the SPL consistent. Finally, the same definitions (product configuration and variation points) often are useful to manage variabilities in other artifacts, such as design and source code. As a consequence, this approach requires the replication of such definitions in different SPL views — when the SPL evolves, changes are propagated throughout many artifacts.

In summary, both approaches 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 widely applied to implement variability at source code. Such techniques are not suitable for modularizing the crosscutting nature of certain features, have poor legibility, and lead to lower maintainability [3]. Consequently, 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 been recommended by the SPL community [26, 20, 14], 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 automatic derivation of product specific artifacts, it is necessary not only to have a more precise definition of each language used to describe product line artifacts and the variability management concern, but also to formalize the weaving processes used to combine them. The PLUSS and PLUC approaches fail in this direction, since Eriksson et al. defines the metamodel of PLUSS notation [16], but do not describe which languages and processes are used for relating use case scenarios to feature models. Likewise, although Fantechi et al. describe the formal semantics of PLUC [17], this approach does not separate variability management from use case scenarios.

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 (using feature models, product configurations, and configuration knowledge) is also applied to other SPL views. Moreover, we believe that our modeling framework, which describes the weaving processes for handling variability management, might be customized for design, implementation, and test artifacts.

3. VARIABILITY AS CROSSCUTTING

Aiming at representing a clear separation between variability management and scenario specification, and also to describe the weaving processes required to compose these views, we propose a modeling framework, that slightly generalizes Masuhara and Kiczales (MK) framework [27], and instantiate it for the use case and product line context. 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 weav-

ing processes take two programs as input, which crosscut each other with respect to the resulting program or computation [27].

Similarly to the MK framework, we represent the semantics of **scenario variability management** as a weaver that takes as input four specifications (*product line use case model*, *feature model*, *product configuration*, and *configuration knowledge*) that crosscut each other with respect to the resulting product specific use case model (Figure 4). Combining these input languages, it is possible to represent the kinds of variability that we are interested in: *optional use cases and scenarios*, *quantified changed scenarios*, and *parameterized* scenarios.

A running example of our approach is presented in Section 3.1. After that, we describe the semantics of our weaving process. For simplicity, it was decomposed in three parts — one weaver process for each kind of variability. The semantics of those weavers (and the meta-model of the input and output languages) are described using the Haskell programming language [24]. This leads to concise weaving processes descriptions (the related source code is available at a web site [2]) and keeps our model close to MK work, where weaving processes are specified in the Scheme programming language. The choice for Haskell was motivated by several factors, such as the existence of mature verification tools (QuickCheck and HUnit), readability, and our previous background in the language.

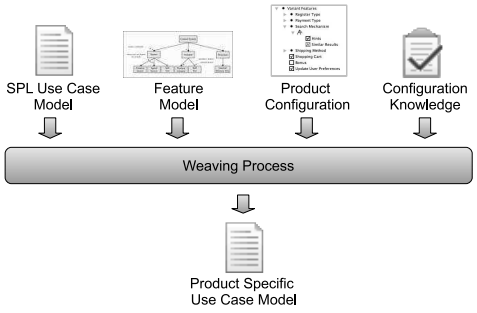

Figure 4: Overview of our weaving process.

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 of our approach based on the eShop Product Line (briefly introduced in Section 2). For this, several artifacts of each input language are described. Then, we present the role of each input language in respect of the weaving process.

3.1.1 SPL use case model

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

Buy a Product: this optional scenario (Figure 5) enables a customer to buy specific goods from an online shopping. It is only available at instances of the product line that are not configured with the *Shopping Cart* and *Bonus* features. This scenarios starts from the IDLE special state (does not extend the behavior of an existing scenario) and finishes at the Step 1M of the *Proceed to Purchase* scenario (which is described a bit later). The clauses *From step* and *To step* are used for describing the possible starting and ending points of execution.

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

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

Figure 5: Buy product scenario.

Buy Products with Shopping Cart and Bonus: this optional scenario (Figure 6) allows the purchasing of products that have been previously added to a customer shopping cart. It changes the behavior of the *Buy a Product* scenario by replacing its first two steps and by introducing the specific behavior required by the *Shopping Cart* and *Bonus* features. This scenario also starts from the IDLE state (*From step* clause) and finishes at Step 1M of *Proceed to Purchase* (*To step* clause). This behavior is required for products that are configured with *Shopping Cart* and *Bonus* features.

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

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

Figure 6: Buy products with shopping cart scenario.

Proceed to Purchase: this mandatory scenario (Figure 7) specifies the common behavior that is required for confirming a purchase. Instances of the product line must be configured with this scenario. Although It can be started either after Step B2 (from *Buy a Product* scenario) or after Step V2 (from *Buy Products Using Shopping Cart* scenario), just one of the paths can be available at a specific product. It is important to notice that the *From step* and *To step* clauses are used for composing, in a quantified way, different scenario configurations without specifying all possible variants in a single artifact (as suggested in the PLUSS notation).

Moreover, notice that a parameter *ShipMethod* is referenced in Step 4M of Figure 7. 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. If a product is configured with the *Economical* and *Fast* ship methods, the final composition of this scenario will present a sentence: *... available ship methods (Economical, Fast).*

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

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

Figure 7: Proceed to purchase scenario.

Search for Products: this mandatory scenario allows the user to search for products. In order to save space, we are only presenting Step S3, which performs a search based on the input criteria (Figure 8). This step is annotated with the mark **[RegisterPreference]**, exposing it as a possible extension point for the behavior of *Register User Preferences*. The same annotation was used in the Step P3 of *Proceed to Purchase* (Figure 7). Such annotations can be referenced in the *from step* and *to step* clauses, reducing problems that might occur by changing step ids.

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

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

Figure 8: Search for products scenario.

Register User Preferences: this optional scenario updates the user preferences based on the buy and search products use cases. Its behavior can be started at each step that is marked with the **[RegisterPreference]** (see the *from step* clause) annotation and is available in products that are configured with the *Update User Preferences* feature.

In this running example, we described several scenarios as being optional or mandatory. It is important to observe that, in our approach, this kind of information is not specified in scenario documents. Actually, it is necessary to consider the relationships between scenarios and features in order to realize which configurations require a specific scenario.

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

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

Figure 9: Register user preferences.

As we said at the beginning of this section, each artifact (feature model, product configuration, configuration knowledge, and use case model) provides a specific contribution to the definition of a SPL member.

3.1.2 Feature model

We have introduced part of this artifact for the eShop product line in Section 2. However, here we present only the features required (Figure 10) for understanding the running example.

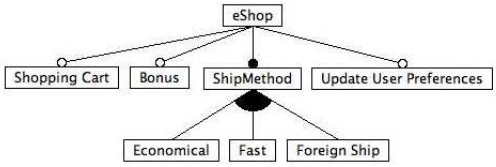

Figure 10: Subset of eShop feature model.

Based on the feature model of Figure 10, the *Shopping Cart*, *Bonus* and *Update User Preferences* features are not required; on the other hand, the *Ship Method* feature is mandatory and all products have to be configured with at least one of its child.

3.1.3 Product configuration

This artifact is used for identifying which features were selected in order to compose a specific member of a product line. Each product configuration should conform to a feature model (the selected features should obey the feature model relationships and constraints). Two possible configurations are presented in Figure 11.

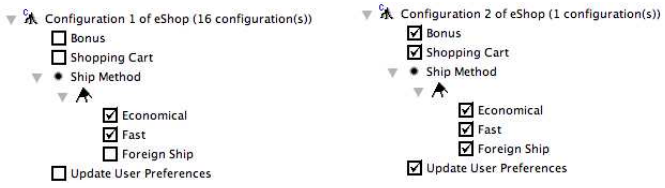

Figure 11: Examples of product configurations.

The first configuration (on the left side of the Figure 11) defines a product that has no support for shopping cart, bonus and preferences update. Additionally, it supports only the economical and fast ship methods. The second configuration selects all possible features. In order to select the required

assets (scenarios, classes and aspects, test cases) for a specific product configuration, it is necessary to relate features to them. This is the role of the configuration knowledge as an input artifact of our weaving process.

3.1.4 Configuration knowledge

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

Table 1 presents a configuration knowledge for the running example, enforcing that

- *Proceed to Purchase* and *Search for Products* are mandatory scenarios, since they are related to the root feature of eShop product line;
- the *Buy a Product* scenario is used in the composition of products that do not have been configured with both *Shopping Cart* and *Bonus* features. Otherwise, if both features were selected for a product, it will be configured with the *Buy Product with Cart* scenario; and
- finally, Table 1 states that *Register User Preference* scenario is not used in composition unless the *Update Preference feature* is selected.

Table 1: eShop configuration knowledge

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

In what follows, we describe a high level view of the weaving process that combines the input languages in order to manage scenario variability. Then, in Section 3.2 we formally present its semantics in terms of our modeling framework.

3.1.5 Weaving process

The weaving process represented in Figure 4 is responsible for performing the following activities:

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

Product derivation activity: This activity takes as input a (valid) product configuration and a configuration knowledge. Each feature expression of the configuration knowledge is checked against the product configuration. If the expression is satisfied, the related scenarios are assembled as the result of this activity. For the running example, Table 2 shows the assembled scenarios for the configurations depicted in Figure 11.

Table 2: Assembled scenarios

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

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

The complete path is a high level representation, which uses the same constructions of the use case model (scenarios), and is illustrated here as a graph, where each node is labeled with a step id. For example, Figure 12 depicts the complete paths for the first and second configurations of our running example. In the left side of the figure, the composition of *Buy a Product* with *Proceed to Purchase* (branch labeled as B1, B2, P1, P2, P3) and *Search for Product* (branch labeled as S1, S2, S3) scenarios are presented. Contrasting, on the right side of the figure, the results of this activity is presented for the second configuration. In this case, steps B1 and B2 have been replaced by steps V1 and V2 (because *Shopping Cart* and *Bonus* features are selected) and the step R1 is introduced after steps P3 and S3 (because *Update User Preferences* is selected in this configuration).

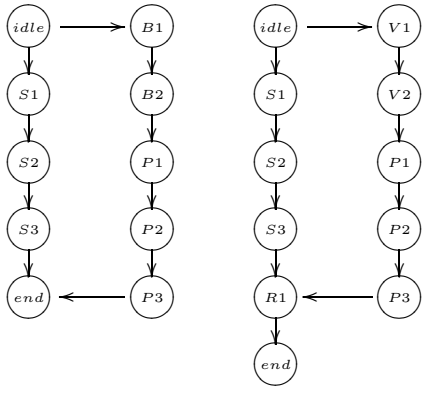

Figure 12: Complete paths represented as a graph

Binding parameters activity: This activity weaves scenarios and product configurations in order to resolve all scenario parameters. For example, step P2 in Figure 7 has a

reference to the *ShipMethod* parameter, whose domain values are defined in the product configuration. For instance, in the first configuration depicted in Figure 11, the parameter *ShipMethod* might assume the values *Economical* or *Fast*. In order to reduce the coupling between scenario specifications and feature model, a mapping is used for relating scenario parameters to features. In fact, this mapping is another input artifact of our modeling framework; but it was not represented in Figure because it was just introduced for avoiding explicit dependences between feature and use case models.

Next, we introduce the modeling framework used to formally describe the weaving processes just presented.

3.2 Modeling Framework

As mentioned before, the semantic of crosscutting, used for representing our variability management weaving process, is based on the Masuhara and Kiczales work [27]. First of all, their requirement for characterizing a mechanism as crosscutting is fulfilled by our approach, since different specifications contribute to the definition of a specific SPL member. As a consequence, due to its crosscutting nature, the modeling framework proposed in [27] is suitable for formalizing variability management compositions.

For simplicity, our weaving process is formally described as being composed by three major weavers: *product derivation weaver*, *scenario composition weaver*, and *bind parameters weaver*. As a customization of MK work, our modeling framework represents each weaver as an 6-tuple (Eq. 1 and Table 3), highlighting the contribution of each input language in the composition processes.

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

Table 3: Modeling framework elements.

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

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

3.2.1 Product derivation weaver

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

For instance, applying this weaver for combining the eShop use case model, feature model , and configuration knowledge with the configuration depicted in right side of Figure 11 will result in the selection of *Buy Products with Cart*, *Proceed to Purchase*, *Search for Products*, ann *Register User Preferences* scenarios.

This weaver is implemented by the function *pdWeaver* (Listing 1) and takes as input a *SPL use case model* (UCM), a *feature model* (FM), a *product configuration* (PC), and a *configuration knowledge* (CK). Initially, this function verifies if the product configuration is a well formed instance of the feature model (*validInstance* function) — if it is not the case, an *InvalidProduct* error is thrown. Then, the IDs of selected scenarios are computed by the *configure* function. This is done by evaluating which feature expressions, defined in the list elements (x:xs) of configuration knowledge, are valid for the specific product instance (*eval* function). Finally, given the resulting list of scenario IDs, the function *retrieveArtifacts* returns the product specific scenarios. As a consequence, we can realize two levels of crosscutting in this weaver. First, the feature model, the product configuration, and the configuration knowledge crosscut each other in order to contribute to the list of valid scenario IDs composition. Then, the resulting list of scenario IDs crosscuts with the use case model for selecting the product specific scenarios.

Listing 1: Product derivation weaver function

```
1  pdWeaver :: UCM -> FM -> PC -> CK -> ScenarioList
2  pdWeaver ucm fm pc ck =
3   if not (validInstance fm pc)
4   then error InvalidProduct
5   else retrieveScenarios ucm (configure pc ck)
6
7  configure :: PC -> CK -> ListOfScenarioId
8  configure pc (CK []) = []
9  configure pc (CK (x:xs)) =
10  if (eval pc (expression x))
11   then (artifacts x) ++ (configure pc (CK xs))
12   else configure pc (CK xs)
```

The model of the *Product Derivation Waver*, in terms of the framework, is showed in Table 4. The *pdWeaver* function is used to argue that the model is realizable and appropriate [27]. We achieve this by matching the model elements to corresponding parameters and auxiliary functions in the implementation code. Therefore, the input languages UCM, FM, CK, and PC are represented as different parameters of the *pdWeaver* function. An instance of the UCM corresponds to the specification of all SPL scenarios. A FM instance is only responsible for declaring the SPL features and the relationships between them; as a consequence, there

is no coupling between FMs and UCMs. Instead, relationships between features and artifacts are documented in the configuration knowledge. Finally, the PC specifies which features were selected for a specific product.

The UCM has a greater importance over the other input languages (UCM_{EFF}), since it declares the parts that compose the product specific scenarios (the output of this weaver process generated by the *pdWeaver* function). These scenarios (UCM_{ID}) are used in the *retriveScenarios* function in order to identify which artifacts will be assembled in the final product.

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

Table 4: Model of Product Derivation

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

3.2.2 Scenario composition weaver

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

As shown in Section 3.1), a variant scenario might refer to steps either in basic or other variant scenarios. In order to compute the complete paths defined by a scenario, we need to compose the events that precede all steps referenced by its *From step clause* (up to the IDLE step), followed by its own steps, and then by all events that follow all of the steps referenced by its *To step clause* (down to the END step)[1].

For instance, consider a product configured with the features *Shopping Cart* and *Bonus*, which requires the *Buy Products with Cart* scenario, and with the feature *Update User Preferences*. Referring to Figure 6, the *Buy Product with Cart* scenario starts from the IDLE state (*From step* clause) and then, after its own flow of events, goes to Step P1 of *Proceed to Purchase* (see the *To step* clause). In a similar way, Figure 9 depicts that *Register User Preferences* scenario starts from any step that is marked with the *RegisterPreferences* annotation (for example, Step P3 of of *Proceed to Purchase*). In this context, the result of applying the composition scenario weaver is a concrete path of execution for this configuration, that can be represented as this sequence of step ids:

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

Note that this sequence still has the *ShipMethod* parameter, referred in Step P2 of *Proceed to Purchase* scenario. The *Binding parameter weaver*, discussed in next section, is responsible for resolving parameters in the final product specification.

Before formalizing the *Scenario composition weaver* in terms of our modeling framework, we first discuss about the abstract representation of scenarios (Listing 2). We omit elements of the use case model abstract syntax that are not required for understanding this weaver. 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 specification in the form of a tuple (user-action x sytem-response), and a list of annotations that can be used to semantically identify the step (avoiding fragile pointcuts). Finally, a reference to a step can be either a reference to a *step id* or to a *step annotation*.

Listing 2: Abstract syntax of scenario artifact

```
1 data Scenario = Scenario id FromStep StepList ToStep
2 data Step = Step Id Action Response Annotations
```

The *Scenario Composition Weaver* is implemented by the *scWeaver* function (Listing 3), which takes as input the product specific use case model (a list of scenarios computed by the previous weaver). The *scWeaver* function computes the complete paths of each scenario by calling, recursively, the *completePaths* function. This function (lines 5-6 in Listing 3) takes as input the product specific use case model (*ucm*) and a specific scenario (*scn*); and returns all complete paths (a list of *step lists*) of *scn*. The function *fromList* (called at line 7) is used to compose all complete paths extracted from the *From step clause*. In a similar way, the function *toList* (called at line 7) is used to compose all complete paths extracted from the *To step clause*. The *match* function (also called at line 7), retrieves all the steps in *ucm* that satisfy all *step references* in *From step* or *To step* clauses. 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 distributed list concatenation.

[1]IDLE and END are predefined steps that represent the *beginning* and the *end points* of a specification.

The model of this weaver is in Table 5. The output (o element of our modeling framework) is the complete paths of the product specific scenarios, computed directly from *scWeaver* function. Therefore, the input language (L) corresponds to the product specific scenarios, related to the *scWeaver* parameters. The join points are modeled as the final scenarios and steps in the output language. They result from the composition of partial scenarios by means of *from steps* and *to steps* clauses (L_{ID}).

The effect of the input languages (L_{EFF}) in the composition process is to combine product specific scenarios that, before this activity, did not define a concrete flow of events. As a consequence, the *match* function plays a fundamental role in this process, retrieving the steps in the use case model that satisfies the *From step* and *To step* clauses.

Table 5: Model of Scenario Composition Weaver

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

After computing the complete paths (using the *scWeaver* function), it is possible to derive another representation for product specific scenarios. This is essentially a *trace model*, since it describe all possible sequences of events specified by the complete paths. This representation is useful for checking, for example, if a non-expected sequence of events is present in a final product, which means that a problem in the composition has occurred. Actually, we used this representation in the verification process of our model (Section 3.3. The *traceModel* function (lines 8 and 9 of Listing 3) is responsible for computing this representation. For instance, the trace model for the the first configuration of our running example is the set of sequences:

$Trace_{C1} =$ {<>, <idle>, <idle, S1>, <idle, S1, S2>,
<idle, S1, S2, S3>, <idle, S1, S2, S3, end>,
<idle, B1>, <idle, B1, B2>, ...,
<idle, B1, B2, P1, P2.ShipMethod, P3, end> }

3.2.3 Bind parameters weaver

This weaver is responsible for the third activity of our variability management process. Parameters are used in scenario specifications in order to create reusable requirements. 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 7 depicts the *Proceed to Purchase* scenario that can be reused for different *ship methods*. Without this parameterized specification, and aiming, for example, at automatically generating a test case suite with a good coverage, it would be necessary to create a specification for each kind of ship method.

This weaver takes into consideration *scenario specifications* and *product configurations*, which defines the domain values of a parameter. Thus, in order to reduce the coupling between scenarios and features, we propose a mapping that relate them. A constraint must be obeyed in this mapping: features related to parameters must be either an **alternative feature** or an **or feature** [19, 15, 14].

The implementation of this weaver consists of calls to the *bpWeaver* function (Listing 4) for each step available in the product specific scenarios or complete paths. This function (lines 1-5 of Listing 4) takes as input a mapping (m), which relates a scenario parameter to a feature; a product configuration (pc), which defines the domain values of parameters (expressed as the feature selection); and a step (s) that may be parameterized. Then, it replaces all parameters from s, returning it as a suitable representation with the corresponding parameter values. Each text between the symbols "<" and ">" (defined in the user action or system response of a step) is treated as a parameter and must be defined in the mapping.

For example, if a product is configured with either *Economical* and *Fast* ship methods, the result of applying this weaver for the Step P2 of the *Proceed to Purchase* scenario will result in the representation (*Economical or Fast*) in each place that the parameter *ShipMethod* is referred.

Listing 4: Bind parameter weaver function

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

Table 6 describes the Bind Parameters model. This weaver just resolves parameters in scenario specifications. Therefore, its output language is also a list of scenarios; but with resolved parameters (the join points). The use case model (UCM) defines the list of scenarios that might be parameterized (UCM_{EFF}). Each step of a scenario (UCM_{ID}), indeed, contributes to the definition of one join point in this weaver. The other contributions come from the configuration knowledge (CK), in the sense that the domain values of a parameter is defined (CK_{EFF}) in the product specific features; and from the mapping (m parameter of the *bind* function) that is used for relating parameters to features.

In what follows, we discuss about some verifications applied to the weaving processes just presented. Such verifications were conduced by applying both *random* and *guided* test cases, and justify even more the choice for a low level design in Haskell programming language.

3.3 Model Verification

The specifications presented in the previous section allow us to verify if the composition processes have desired properties or behavior. Aiming at doing that, we applied two techniques for testing Haskell programs: unit tests; and formal specifications for checking properties of our modeling framework.

Unit tests were developed using the HUnit library [22]. Sim-

Listing 3: Scenario composition weaver function

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

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

traceModel [] = [[]]
traceModel (x:xs) = [] : (x) ^ (traceModel (xs))
```

Table 6: Model of Bind Parameters Weaver

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

ilar to other *xUnit* tools, it requires well defined input data and expected results. After that, it is possible to check if a call to a *function under test*, sending the previous defined input data as argument, yields a value equal to the expected results. For instance, we applied unit tests for checking if the composition process for the product configurations depicted in Figure 11 yields expected traces. Therefore, the trace model notation discussed in a previous section plays an important role in our verification process. In order to perform such kind of checking, we first implemented a *refine* function (Listing 5), which checks if all sequences of traces in the *model under test* (mut) are present in the expected results — named here as reference model.

Listing 5: The *traceRefinement* function

```
refine :: TraceModel -> TraceModel -> Bool
refine referenceModel mut =
 and [exists (x referenceModel) | x <- mut]
```

Then, we defined expected traces (such as *data01* and *data02* in Listing 6) for both configurations of Figure 11, considering the complete paths showed in Figure 12. Additionally, two trace models (*tm01* and *tm02*), computed by the composition process, were defined as input data. Notice that the input trace models were computed varying just the products configurations (*pc01* and *pc02*). Finally, test cases (such as tc01 and tc04) were developed for verifying expected and non-expected traces in the resulting composed models.

Listing 6: Unit test for composition process

```
data01 = [[], [idle], [idle, 1M], ...
 [idle,1M, 2M, 3M, 4M, 5M, end]]

data02 = [[],[idle], [idle, V1], ...
 [idle, V1, V2, 3M, 4M, 5M, end]]

tm01 = computeTraces (fm01 pc01 ck01 ucm01)
tm02 = computeTraces (fm01 pc02 ck01 ucm01)
-- expected traces for configuration 01
tc01 =
 TestCase (assertBool (refine tm01 data01 tm01))

-- non-expected traces for configuration 02
tc04 =
 TestCase (assertBool (not (refine data01 tm02)))
```

Based on the examples just presented, unit tests are useful for verifying the presence of errors under well defined test cases (described by the input data and expected results). A complementary approach consists in checking function properties based on formal specifications. In order to perform this kind of verification, we use the *QuickCheck* library [10], which provides an *embedded specific language* that is used to specify properties of Haskell programs; and several functions for checking formal properties and for randomly generating input data.

This second verification approach is also suitable in our context since several properties related to each input model (or to the compositions between them) should be obeyed. Applying just unit tests for checking these properties may require a huge effort for designing and implementing input data and expected results for each interesting test scenario. On the other hand, this formal technique requires the specification of properties that a *function under test* should hold. These properties are then checked against randomly input data.

We applied this verification technique for checking several properties of feature modeling (briefly introduced in Section 2). For example, an alternative feature requires exactly one sub feature to be selected in a product configuration. Using the *QuickCheck* library, this property can be expressed as the function *prop_ExactlyOneSelected* in Listing 7. It takes two feature as parameters and returns a property to be checked. The first parameter (fm) should be an alternative feature (from a feature model). The second

one (fc), instead, is a corresponding feature from the product configuration. If the number of children of *fc* is different than one (the expression before the implies symbol), it is expected that a call to the *existError* function will return *True*. After specifying this kind of property, it is possible to verify if they hold for different input data. The input size and the algorithm used for generating input data can be configured by the tester.

We also performed this kind of verification for checking properties related to the weaving process. For instance, we defined properties for checking if all scenario parameters are related exclusively to *alternativeFeatures* or *orFeatures*. Such constraint was discussed in the previous section.

Both verification approaches were important to improve the confidence of our models. Actually, several unit tests and properties revealed to us interesting case; we were not initially concerned with some of them. To our knowledge, no existing work for representing variability management in scenarios apply these level of verifications.

In the next section we present an evaluation of our approach based on the specification of SPLs in different domains.

4. EVALUATION

We have applied our approach to three SPLs: the eShop Product Line, introduced in Section 2; the **Pedagogical Product Line (PPL)** [28], which was proposed for learning about and experimenting with software product lines and focus on the arcade game domain; and the **Multimedia Message Product Line (MMS-PL)**, which allows the assembling of specific products for creating, sending, and receiving multimedia messages (MMS) in mobile phones.

Based on the last application of our approach, we noticed the benefits of a clear separation between variability management and scenario specification [8], compared our approach with the PLUC and PLUSS techniques. This comparison uses Design Structure Matrices (DSMs), a suite of metrics for quantifying modularity and complexity of specifications, and observations of the effort required to introduce SPL increments (such as new variants or products). It is important to note that we did not formalize variability management as crosscutting in our previous work [8]; we just report on the benefits related to the *separation of concerns* (SoC) claimed in this paper.

4.1 DSM Analysis

For now, let us reproduce the DSMs to understand the benefits of SoC in variability management. DSMs is an interesting and simple tool for visualizing dependences between design decisions [6]. Such decisions appear in the rows and columns of a matrix. We can identify a dependency by observing the columns in a given row [6]. For example, the first row in Figure 13 indicates that the task of creating the feature model depends on the task of creating the use case model. The first can not be independently performed after the second. As presented in Section 2, the PLUC approach describes product instances and variability space at specific sections of use cases. Therefore, it is not possible to evolve variability management (introducing new features, products, or relations between features and artifacts) without reviewing the use case model. This is expressed in the non modular DSM of Figure 13, which depicts cyclical dependences between use cases and variability management artifacts.

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

Figure 13: DSM Analysis of PLUC

Our approach reduces the dependences between variability management and scenario specifications (Figure 14). For instance, changes in feature model or new definitions of products do not require changes in the use case model. This clear separation is also necessary in source code, as claimed in [3, 5], and might be required in other artifacts too. Notice that the proposed mapping artifact, used to relate use cases parameters and features (Section 3.2.3), can be considered as a design rule, since it primarily aims at decoupling use cases and features.

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

Figure 14: DSM Analysis of the proposed approach

4.2 Quantitative Analysis

For confirming the observations derived from the DSM analysis, we applied the metric suite for the *MMS* and *Pedagogical* product lines. We compared our specification of *MMS* product line to the specifications we wrote for both PLUC and PLUSS techniques [2]. In a similar way, we compared our specification of the *Pedagogical* product line to a specification that was sent to us by the authors of the PLUSS approach.

The metric suite, used in what follows, was adapted from [18] for both product line and use cases contexts. It quantifies feature modularity and use case complexity. The proposed modularity metrics quantify three types of relations involving features and use cases. First, *Feature Diffusion over Use Cases* (FDU) is used for quantifying how many use cases are affected by a specific feature. On the other hand, *Number of Features per Use Case* (NFU) is used for quantifying how many features are tangled within a specific use case. We assume that each use case should focus in its primary goal, although several features might be related to the primary goal of a use case. Finally, we applied the metric *Feature Diffusion over Scenarios* (FDS) in order to quantify how many internal use case members (scenarios) are necessary for the materialization of a specific feature.

Moreover, we used three metrics related to complexity. The first one, *Vocabulary Size*, quantifies the number of use cases (VSU) and scenarios (VSS) required by each of the evaluated approaches. The second one, *Steps of Specification* (SS), is related to the size of each scenario and identifies

Listing 7: Example of QuickCheck property

```
1 prop_ExactlyOneSelected :: Feature -> Feature -> QuickCheck.Property
2 prop_ExactlyOneSelected fm fc = (not (length (children fc) == 1)) ==>
3  existError (checkAlternativeFeature fm fc)
```

how many pairs *User action* x *System response* compose a specific scenario. Additionally, we also relate modularity to complexity by applying *Features and Steps of Specification* (FSS), which counts the number of steps of specification whose main purpose is to describe the behavior of a feature. A complete description of these metrics can be found elsewhere [8]. Next we present the quantitative analysis of the *MMS* and *Pedagogical* product lines using these metrics.

4.2.1 MMS Product Line

As explained earlier, the MMS product line, which is a real case study from Motorola phones, enables the customization of multimedia message applications. The primary goal of each one of these applications is to create and send messages with embedded multimedia content (image, audio, video) [8]. We have specified the MMS product line using three techniques: our notation, PLUC, and PLUSS approaches. After that, we evaluated these specifications observing the metric suite just presented. The results of this evaluation are summarized in Table 7.

Table 7: MMS quantitative evaluation

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

Since PLUC and PLUSS do not allow a scenario to crosscut other scenarios in different use cases, it is difficult to modularize features into use cases. The result is that, when comparing to the crosscutting approach, features, on the average, are more diffused (FDU, FDS, and FSS metrics) and use cases are less concise (NFU) in these approaches (Table 7). The crosscutting approach, in contrast, allows the composition of scenarios through *from steps* and *to steps* clauses.

The lack of crosscutting mechanisms for composing scenarios of different use cases also results in greater complexity, when observing the *Steps of Specification* metric. Although our approach resulted in a greater number of use cases, we improve the specification reuse, since scenarios that crosscut different use cases can be specified without duplication. For instance, consider the *Structure Data Operation* feature which presents this crosscutting nature. As we can realize observing Figure 15, by applying our approach we can reduce the diffusion over scenario of such features.

4.2.2 Pedagogical Product Line

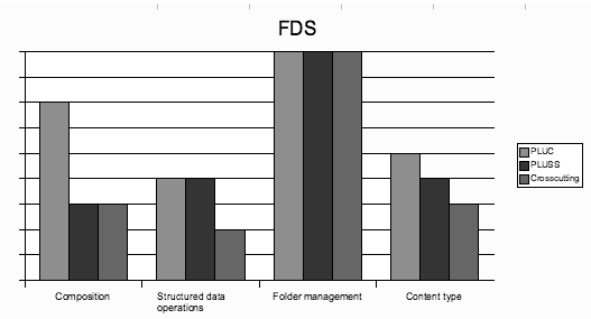

Figure 15: Relative FDS for evaluated techniques

We compared our approach against two specifications of the Pedagogical Product Line (PPL): the original one [28], proposed by the Software Engineering Institute, and a specification that was sent to us by the authors of the PLUSS technique.

The original specification of PPL is already well modularized, since its features, in general, are not crosscutting among different use cases (see modularity metrics in Table 8). Moreover, another characteristic of PPL is that several features are related to qualities that do not cause effect into use case specifications.

Table 8: PPL quantitative evaluation

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

Still in this context, our approach also achieves some improvements in the resulting specifications. The main factor for these improvements in the *Pedagogical* product line was the error handling modularization. In both SEI and PLUSS specifications, the *error handling* concern was spread into several use cases.

However, by applying our approach, all behavior related to the *error handling* concern is specified in a single use case. The composition of *error handling* with the basic scenarios was done by means of annotations attributed to the corresponding steps. For instance, Figure 16 depicts just one scenario for error handling: raising an error when there is no space available.

```
Description:  There is no available space in file system.
From step:  [CatchFileException]
To step:  END
```

Id	User Action	System State	System Response
E1		There is not enough space to save the file.	Raise the Disk is Full exception. The arcade game application is finished.

Figure 16: Scenario of error handle use case.

Notice that this scenario can be started from any step that has been marked with the *CatchFileException* annotation (see the *From step* clause). Several features of the PPL need to save information in the file system. Therefore, both in the SEI and PLUSS specifications of the *Pedagogical* product line, several use cases were specified with scenarios for handling this kind of exception.

As a consequence, we achieve a reduction (almost 20%) in the number of specification steps (SS in Table 8) when comparing to the PLUSS approach. It is important to notice that this reduction of size does not compromise the requirement coverage; but actually it represents an improvement in the specification reuse.

For concluding our Pedagogical product line analysis, we can realize, based on Table 8, that our approach achieved real benefits only in the complexity metrics. This result comes from the non crosscutting nature of PPL features. Comparing to the MMS product line results, we argue that the benefits of applying our approach should be greater in contexts where features can not be well modularized using existing techniques.

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 [30]. 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 [23]. 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.

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 [29, 13]. 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 (metamodel) 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 [9]. Such approach is based on natural language processing. Using our scenario composition weaver (Section 3.2.2), a scenario composition can be represented using references to *step ids* 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 3.

6. CONCLUSIONS AND FUTURE WORK

In this paper we formally described variability management as a crosscutting mechanism, considering the contribution of different input languages that crosscut each other for deriving specific members of a product line.

We applied this notion of variability management in the context of use case scenarios, achieving several benefits. First, applying our approach resulted in a clear separation of concerns between variability and scenario specifications, allowing both representations to evolve independently. Second, we achieved a reduction in the size of specifications by composing scenarios in a quantified way. Finally, the formal specification allowed us to perform several automatic verification in the composition process. This is an interesting characteristic of our approach, since it can be used for finding inconsistencies in the final products.

Although our modeling framework was instantiated for representing scenario variabilities, we believe that it could be applied in other SPL artifacts. Particularly, optional and parameterized artifacts are also relevant for non-functional requirements and test cases. Additionally, our notion of crosscutting was based on a work that states *what* defines a source code technology as supporting crosscutting modularity. Therefore, we argue that representing variabilities in source code, using our modeling framework, is straightforward.

As future work, we would like to apply our notion of variabil-

ity management in order to identify relationships between variabilities at different SPL artifacts (in a first moment, relationships between requirements and 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 formally represented.

7. 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 *1st Workshop on Aspect-Oriented Product Line Engineering*, Portland, USA, Oct 2006.

[4] K. P. Andreas Reuys, Erik Kamsties and S. Reis. Model-based system testing of software product families. In *Proceedings of 17st International Conference on Advanced Information Systems Engineering (CAiSE'05)*, Porto, Portugal, 2005.

[5] S. Apel, T. Leich, and G. Saake. Aspectual mixin layers: Aspects and features in concert. In *Proceedings of International Conference on Software Engineering*, 2006.

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

[7] 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.

[8] R. Bonifácio, P. Borba, and S. Soares. On the benefits of variability management as crosscutting. In *Early Aspects 2008, affiliated with AOSD'08*, Brussels, Belgium, mar 2008.

[9] 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.

[10] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *International Conference in Functional Programming (ICSP'2000)*, pages 268–279, 2000.

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

[12] J. Conejero, J. Hernandez, A. Moreira, and J. Araújo. Discovering volatile and aspectual requirements using a crosscutting pattern. In *15th IEEE International Requirements Engineering Conference*, 2007.

[13] 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.

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

[15] 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.

[16] M. Eriksson, J. Borstler, and K. Borg. The pluss approach - domain modeling with features, use cases and use case realizations. In *SPLC'05*, 2005.

[17] 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.

[18] A. Garcia, C. SantAnna, E. Figueiredo, U. Kulesza, and C. Lucena. Modularizing design patterns with aspects: A quantitative study. In *Transactions on Aspect-Oriented Software Development*. Springer, 2005.

[19] 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.

[20] J. Greenfield, K. Short, S. Cook, and S. Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, 2004.

[21] 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.

[22] D. Herington. Hunit 1.0 user's guide. online: http://hunit.sourceforge.net/HUnit-1.0/Guide.html (2008).

[23] J. Hunt and J. McGregor. Implementing a variation point: A pattern language. In *Variability Management Workshop, SPLC'06*, 2006.

[24] P. Jones. *Haskell 98 Language and Libraries: the Revised Report*. Cambridge University Press, 2003.

[25] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature oriented domain analysis: feasibility study, 1990.

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

[27] H. Masuhara and G. Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In *ECOOP: Proceedings of the European Conference on Object-Oriented Programming*, 2003.

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

[29] N. Noda and T. Kishi. An aspect-oriented modeling mechanism based on state diagrams. In *AOM'06*, Genova, Italy, 2006.

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

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

[32] G. Sousa, S. Soares, P. Borba, and J. Castro. Separation of crosscutting concerns from requirements to design: Adapting the use case driven approach. In B. Tekinerdoğan, P. Clements, A. Moreira, and J. Araújo, editors, *Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design*, 2004.