an approach → o framework é outra coisa, dá a semântica da approach

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

and that artifacts such as FM and CK affect...

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 occurs at different SPL views (requirements, design, implementation, and test). In this paper, we present ~~a framework~~ for modeling use case scenario variability mechanisms, 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 features often affect different points of each SPL view. Additionally, we believe that our proposed framework might be customized to describe variability mechanisms in other SPL artifacts, being a contribution for automatic product generation and traceability.

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 [28]. However, variability management, due to its inherent crosscutting nature, is a common challenge in software product line (SPL) adoption [9, 28]. 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). ~~In this case~~, 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 [28].

So

não é por aí

Several authors have proposed the use of *aspect-oriented* mechanisms in order to better modularize variability management at source code [3, 13]. However, to our knowledge, no attempt has been done to verify which are the crosscutting mechanisms that could be applied for managing variabilities at requirements, design, and test artifacts. Actually, current techniques for scenario variability management [20, 5, 14] do not present a clear separation between variability management and scenario specification. For instance, supposing that product variants are tangled with use case scenarios, if one variant is removed from the product line, it would be necessary to change all related scenarios. In summary, it is difficult to evolve both representations.

Another problem is the lack of a formal representation for the weaving processes used to derive product specific scenarios. This is not suitable for current SPL generative practices [25]; 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 basic and variant flows are correct. Although the semantic composition of PLUC (Product Line Use Case) [5] is defined in [15], such approach does not separate scenario specification from variability management. ~~In order to explain in more details the problems addressed by this work, a motivating example is~~ as presented in Section 2.

In this paper, we describe a framework for modeling the composition process of scenario variabilities (Section 3). Such framework aims to: (1) represent a clear separation between variability management and scenario specification; and (2) specify how to weave these representations in order to generate specific scenarios for a SPL member. Therefore, the main contributions of this work are

- A formal characterization of variability management as a crosscutting concern and, in this way, enforce that it must be represented as an independent view of the SPL. Although this work focus on requirement artifacts, more specifically use case scenarios, we argue that such separation is also valid for other SPL views. Actually, it has already been claimed for source code [3, 13].

- A framework for modeling the composition process of scenario variability mechanisms. This framework gives a basis for describing variability mechanisms, allowing a better understanding of each mechanism. In this work, such framework is used for modeling the semantics of scenario variation mechanisms, but it might be customized for other SPL views.

- Describe three scenario variation mechanisms 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 modeling framework is based on Masuhara and Kiczales work [26], another contribution of this paper is that we have applied their definition about *what* characterize a mechanism as crosscutting in the domain of variability management. Based on their view of crosscutting, we can reason about variability management as a crosscutting concern that weaves different input specifications in order to derive a product specific member.

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

2. MOTIVATING EXAMPLE

In order to customize specific products, by selecting a valid feature configuration, variant new points must be represented in the product line artifacts. Several variant new notations have been proposed for use case scenarios, like Product Line Use Cases (PLUC) [5] and Product Line Use Case Modeling for Systems and Software Engineering (PLUSS) [14]. 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* [29] 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 [29]. Here, we consider extra features, such as *Update User Preferences*, which ~~that~~, based on the user historical data of searches and purchases, updates user preferences. Figure 1 presents part of the *eShop* feature model [18, 11, 24]. ~~As a brief introduction about the feature model notation,~~ the relationships between a parent and its child are ~~categorized~~ re- as: *Optional* (features that might not be select ~~in a~~ specific product), *Mandatory* (feature that must be selected, if the parent is also selected), *Or* (one or more subfeatures might be selected), and *Alternative* (exactly one subfeature must be selected).

for each product

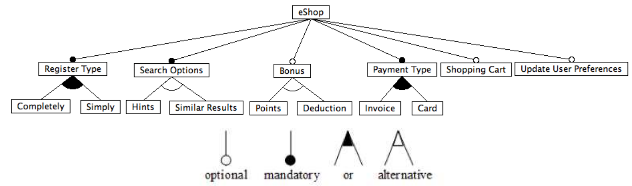

Figure 1: eShop feature model.

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.

explica que para entender estas relações, e com

Tem que olhar então o artefato
não esse

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.

product
the behavior of

Following this approach, it is hard to understand a specific ~~configuration~~ because all possible variants are described in the same artifact. Also, such tangling between variant representation and scenario specification results in maintainability issues: introducing a new product variant might require changes in several points of existing artifacts. For example, including a *B2B Integration* feature, which allows the integration between partners in order to share their warehouses, might change the specification of 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 can be spread among several specifications and each specification might describe several variants, the efforts needed to understand and evolve the product line might increase.

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

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, instead, is not configured with such features.

Since the values of alternative and optional variation points are computed based on the defined products, instead of specific feature, the inclusion of a new member in the product line might require a deeply review of variation points. Moreover, since the variation points and the product definitions are spread among several scenario specifications, it is hard and time consuming 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 - if the SPL evolved, changes would be propagated throughout many artifacts.

In summary, both techniques rely on simple composition techniques: filtering variant steps in scenarios or syntactic changes of tag values based on product definition. In this sense, they are similar to conditional compilation techniques, which have been applied to implement variability at source code. Such techniques are not suitable for modularizing the crosscutting nature of certain features, has poor legibility, and leads to lower maintainability [3]. 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 being recommended by the SPL community [25, 19, 11], in such way that the combination of generative techniques, aspect oriented programming, and software product line should be further investigated. In this case, in order to support the automatically derivation of product specific artifacts, it is necessary not only a more precise definition of each language used to describe product line artifacts and the variability management concern, but also the weave processes used to combine them. PLUSS and PLUC approaches fail in this direction, since Eriksson et al., although present the metamodel of PLUSS notation [14], do not describe which languages and processes are used for relating use case scenarios to feature models. Likewise, besides Fantechi et al. describe the formal semantics of PLUC [15], this approach does not separate variability management from use case scenarios.

The next section describes our approach that considers scenario variability as a composition of different artifacts. Although in this paper we are focus on use case scenarios, the idea of separating product line artifacts from variability management is also applied to other SPL views. Moreover, we believe that our modeling framework, which describes the weaving processes for handle variability management, might be customized for design, implementation, and test artifacts.

ing

do search/replace no texto todo

3. VARIABILITY AS CROSSCUTTING

Aiming to represent a clear separation between variability management and scenario specification, and also to describe the weave processes required to compose those views, we ~~are~~ proposing a modeling framework that ~~is a customization of~~ ~~the~~ Masuhara and Kiczales (MK) work [26]. The goal of MK framework is to explain how different *aspect-oriented* technologies support crosscutting modularity. Each technology is modeled as a three-part description: the related weave processes take two programs as input, which crosscut each other with respect to the resulting program or computation [26].

ing

framework

Similarly to the MK framework, we represent the semantics of **scenario variability management** as a weaver that takes four specifications as input (*product line use case model*, *feature model*, *product configuration*, and *configura-*

slightly generalizes

and instantiates it for the use case and PLine context melhor 1B, 2B, ..., B de Buy

dá a impressão que é só um, coloca no plural.

tion 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.

that This leads to (parts

A running example of our approach is presented in Section 3.1. After, we describe the semantics of our weaving process. For simplicity, it was decomposed in three ~~subcomponents -~~ one weaver process for each kind of variability. The semantics of those weavers (and the meta-model of the input and output languages) are described using the Haskell programming language [23]. ~~Such decision allows the execution and testing of concise weaving processes descriptions~~ where (~~all~~ the related source code is available at SPL a web site [2]) and keeps our model close to MK work, ~~which specifies their~~ weaving processes ~~using~~ the Scheme programming language.

are specified in the

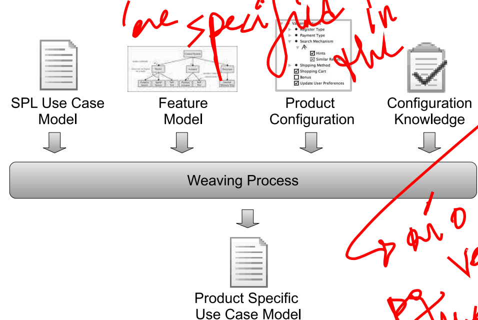

Figure 4: Overview of the modeling framework.

só o com pergunta pq você não usou scheme. Fica mais conciso que em scheme.

evitar esta referência ao SBES

3.1 Running example

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

products of our approach

3.1.1 SPL use case model

This artifact defines a set of scenarios that might be used to describe possible variants of the product line. Although not being directly concerned with variability management, some scenarios might be optional, might have parameters, or might change the behavior of other scenarios. Based on the notation proposed in [7], a use case scenario corresponds to a sequence of steps (a tuple of *User Action* x *System State* x *System Response*). A use case defines a set of scenarios; and a use case model ~~instead~~ defines a set of use cases. In this running example, we ~~are~~ considering the following scenarios:

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

resolve problema mas invade a base!

extension

```
Description: Basic version of Buy Products
From step: IDLE
To step: END
```

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

Figure 5: Basic version of Buy Products.

Notice that a parameter *ShipMethod* is referenced in Step 4M of Figure 5. The use of this parameter (notation also supported in PLUSS and PLUC) allows the reuse of this specification for different kinds of *ship method* configurations. For example, when instantiated with Fast and Economical ...

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

```
Description: Extended version of Buy Products
From step: IDLE
To step: 3M
```

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 shopping cart.
V2	Select the confirm option.	Request bonus and payment information.

nos exemplos não se usa State! e como estas tabelas já aparecem antes n faz sentido explicá-las aqui

Figure 6: ~~Extended~~ version of Buy Products.

Search for Products: this scenario allows the user to search for products. In order to save space, we are only presenting the Step 3S, which performs a search based on the input criteria (Figure 7). This step is annotated with the mark **[RegisterPreference]**, exposing it as a possible point ~~that~~ the behavior of *Register User Preferences* ~~might starts~~. The same annotation was used in the Step 5M of *Buy Products Basic Version*. Such annotations can be referenced in the *from step* and *to step* clauses, reducing the problem of fragile pointcuts [8].

for

já falou em aspectos antes? se não, fica meio sem contexto isso aqui

como o que discutiremos a seguir...
(fica legal pôr link entre os passos)

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

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

ok, mas deixa claro que você deixou isso dos templates, ok? diz que requer outros artefatos

Figure 7: Search for Products scenario.

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

fica depois

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

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

o ideal seria dizer que volta para o passo seguinte, não é?

Figure 8: Register user preferences.

3.1.2 Feature model

We have introduced part of this artifact for the eShop product line in Section 2. However, here we ~~are going to~~ now present only the features required (Figure 9) for understanding the running example.

e se for só cart ou só bonus?

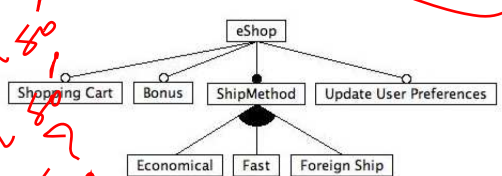

Figure 9: Subset of eShop feature model.

Based on the feature model of Figure 9, the *Shopping Cart*, *Bonus* and *Update User Preferences* features are not required; on the other hand, the feature *Ship Method* is mandatory and all ~~specific~~ products ~~might~~ be configured with at least one of its child.

being

have to be

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 ~~be~~ conform to a feature model (the selected features should obey ~~to~~ the feature model relationships and constraints). Two possible configurations are presented in Figure 10.

The first configuration (on the left side of the Figure 10) defines a product that has no support for shopping cart, bonus

a instanciação de cenários ocorre aí?

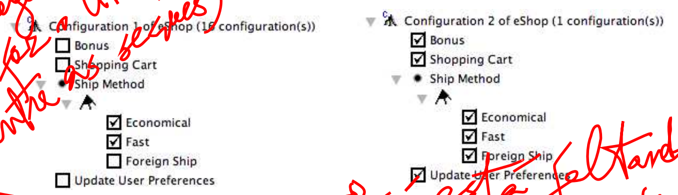

Figure 10: Examples of product configurations.

só está faltando um pouco de link entre as seções

and preferences update. Additionally, it supports only the economical and fast ship methods. The second configuration selects all possible features.

3.1.4 Configuration knowledge

This artifact is used for relating feature expressions to artifacts that must be assembled in a given product. Such artifacts allow, during ~~the~~ product engineering ~~phase~~, the automatic ~~ally~~ selection of assets that are required for a specific product configuration. Table 1 presents the configuration knowledge for the running example, enforcing that ~~,~~ if *Shopping Cart* and *Bonus* features are **not** selected, the basic version of *Buy Product* scenario will be assembled; otherwise, the extended version of the same scenario will be assembled; and the *Register User Preferences* scenario will be assembled only if the *Update User Preferences* feature is selected.

aqui teria que ser a Basic e Extended

Table 1: eShop configuration knowledge

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

furado! falar comigo

3.1.5 Weaving process

After presenting input artifacts for the running example, we ~~are ready to~~ describe the weaving process that combines the input languages in order to manage scenario variabilities. In the next section we present, more precisely, the semantics of its ~~components using a low level design. The weaving process is composed by four activities, although the last one is optional:~~

subprocesses, which perform 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 ~~is~~ conforms to the relationship cardinalities and constraints of the feature model), the process might proceed. ~~Otherwise, an error should be reported.~~

Product derivation activity: This activity takes as input a (valid) product configuration and a configuration knowledge. ~~Then~~ Then, 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 10.

isso tem que aparecer nas atividades anteriores

Basic tb!

Table 2: Assembled scenarios

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

seria melhor usar o mesmo fonte dos templates

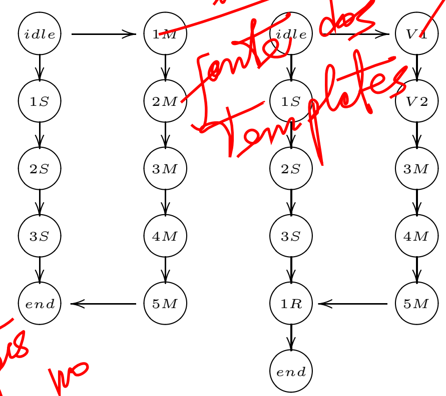

Figure 11: Complete paths represented as a graph

melhor escolher um: se for o caso, mas conclusões vc fala no outro

Scenario composition activity: This activity is responsible for composing the scenarios assembled for a specific product configuration. The resulting scenarios of the previous activity, which crosscut each other based on the *from step* and *to step clauses*, are woven. The result is either a use case model with complete paths (all *from step* and *to step* clauses are resolved) or a trace model (a set of all valid sequences of events extracted from the complete paths).

is illustrated here by

The complete path is a high level representation, which uses the same constructions of the use case model (scenarios), and ~~can be better understood using~~ a graph notation, where each node is labeled with a step id. For example, Figure 11 depicts the complete paths for the first and second configurations of our running example. In the left side of the figure, the basic versions of search for product (branch labeled as 1S, 2S, 3S) and buy product (branch labeled as 1M, 2M, ..., 5M) scenarios are presented. ~~Instead~~, on the right side of the figure, the extended versions of these scenarios are presented. In this case, steps 1M and 2M have been replaced by steps V1 and V2 (because *Shopping Cart* and *Bonus* features are selected) and the step 1R is introduced after steps 5M and 3S (because *Update User Preferences* is selected in this configuration).

containing

On the other hand hum!?

~~Instead~~, the trace model can be seen as a low level representation of the use case model. Such notation has a well defined semantic and might be used for model checking and test case generation. Such applications of the trace model are beyond the scope of this paper. More information can be found elsewhere[21, 30, 16]. Here, the trace model is useful for implementing the last activity of our weave process, binding parameters, and represents all possible sequences of events in a specific product configuration.

complete path também

ing

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

realizar esta atividade

$Trace_{C1}$ = {<>, <idle>, <idle, 1S>, <idle, 1S, 2S>, <idle, 1S, 2S, 3S>, <idle, 1S, 2S, 3S, end>, <idle, 1M>, <idle, 1M, 2M>, . . . , <idle, 1M, 2M, 3M, 4M.ShipMethod, 5M, end> }

Binding parameters activity: This optional activity takes as input the assembled scenarios and the product configuration and generates a trace model resolving all parameters referenced by the steps. For example, step 4M in Figure 5 has a reference to the parameter *ShipMethod*. The parameter values are defined in the product configuration.

esse não é um bom termo; se o caso de uso tem parâmetros tem que

sem ele não da para implementar parâmetros?! estranho.

Falar comigo!

and should

For instance, in the first configuration depicted in Figure 10, the parameter *ShipMethod* ~~might~~ assume the values *Economical* or *Fast*. ~~In order to reduce the coupling between scenario specifications and feature model, a mapping (or environment) is used for relating scenario parameters to features.~~ For each trace that contains a parameterized event (or step), this activity creates a new trace for all of the possible parameter values. Consequently, resolving parameters for the trace $< idle, 1M, 2M, 3M, 4M.ShipMethod >$ results in the following sequences:

pelo que conversamos no telefone seria 4M.Economical or Fast

isso parece ser um detalhe da implementação em Haskell não cabe aqui

<idle,1M,2M,3M,4M.Economical>,
<idle,1M,2M,3M,4M.Fast>,

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

just

3.2 Modeling Framework

As ~~presented~~ before, we are proposing, based on the Masuhara and Kiczales work [26], a crosscutting variability management approach. 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. Additionally, due to its crosscutting nature, the modeling framework proposed in [26] is suitable for formalizing variability management compositions. Our weaving process is modularized in three major components, formally described in sections ~~3.2.1, 3.2.2, 3.2.3~~. Actually, each of this components is also a weaver. As a customization of MK work, our modeling framework, used to describe these weavers, is an 7-tuple (Eq. 1 and Table 3) that highlights the importance of the composition process languages.

simplificar texto

melhor reescrever este parágrafo acima

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

Each component is responsible for realizing one variability mechanism. We represent them by filling in the seven parameters of our modeling framework and by stating how elements of the weaver implementation correspond to elements of the model.

weavers in

reescrever tb

falar comigo

um bolo tudo, só com todas as linguagens, misturado?

Table 3: Modeling framework elements.

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

Não! conjuntos indexados

implements

3.2.1 Product derivation weaver

This weaver is responsible for selecting artifacts based on specific product configurations. As a consequence, it ~~is worried about~~ 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. In this paper, we focus only in the selection of scenarios that should be assembled in specific instances of the SPL. However, this weaver often is required for managing variabilities in other kinds of artifacts. 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 10 will result, beside others, in the selection of the extended version of *Buy Products* scenario (Figure 6).

?

This weaver, and is implemented by the function *pdWeaver* (Listing 1), 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 ~~required~~ selected scenarios are computed by the *configure* function. This is done by evaluating which feature expressions (list *x:xs*), in the 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.

The model of the *Product Derivation Weaver*, in terms of the framework, is showed in Table 4. The *pdWeaver* function is used to argue that the model is realizable and appropriate [26]. 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.

reforçar onde está o crosscutting aí

confuso

Listing 1: The *configure weaver* function

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

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

o que pode facilitar o entendimento é mostrar como parte de CK de Tabela 1 é representado por esta lista

parts that compose the

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

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

The product derivation weaver is not parameterized. So, the *meta* element of the modeling framework is not considered in this case.

Includes on (on selects)

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
PC_{ID}	Product specific feature selection
UCM_{EFF}	Provides declaration of scenarios
FM_{EFF}	Checks if a SPL instance is well formed
CK_{EFF}	~~Defines the required artifacts~~ scenarios
PC_{EFF}	Specifies the configuration of a product
UCM_{MOD}	Scenario
FM_{MOD}	Feature
CK_{MOD}	Each pair ($feature\ expression, artifact\ list$)
PC_{MOD}	Feature
$meta$	-

ou lista de scenario IDs?

? não seria triggers scenario selection

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 [7] and is applied whenever a use case scenario supports different paths of execution, based on the product configuration. This mechanism takes as input the product specific use case model (a list of scenarios). Each flow of events (usually a partial specification) ~~must be~~ composed in order to generate a concrete scenario.

nem precisa ser variant, certo?

is then

As shown in uniformizar a terminologia

up to

A variant scenario might refer to steps either in basic or other alternative scenarios. In order to compute the complete paths defined by a scenario, we need to compose, ~~recursively,~~ the events that precede all ~~of the~~ steps referenced by its *from step clause* (~~until~~ the IDLE state), ~~plus~~ its own steps, ~~plus~~ all ~~the~~ events that follow ~~all of~~ the steps referenced by its *to step clause* (~~until~~ the END state)[1].

down to

For instance, consider a product configured with the features *Shopping Cart* and *Bonus*, which requires the extended version of *Buy Product* scenario, and with the feature *Update User Preferecens*. Referring to Figure 6, the extended version of *Buy Product* scenario starts from the IDLE state (*from step* clause) and then, after its own flow of events, goes back to Step 3M of *Buy Product* basic version (*to step* clause). In a similar way, Figure 8 depicts that *Register User Preferences* scenario starts from any step that is marked with the *RegisterPreferences* annotation (for example, Step 5M of the basic version of *Buy Product*). 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 ~~the~~ sequence of step ids:

<IDLE, V1, V2, 3M, 4M.ShipMethod, 5M, 1R, END>.

that

nova parágrafo

Note that this sequence still has the *ShipMethod* parameter, referred in Step 4M of *Buy Product* basic flow. Listing 2 presents the abstract representation of the use case model, use case, and scenario artifacts. Some elements of the abstract syntax, ~~which~~ are not required for understanding this weaver, ~~are not presented in Listing 2~~. Briefly, a use case model has a name and a list of use cases. A use case has an id, a name, a description, and a list of scenarios (~~that~~ defines ~~its~~ basic, alternative, and exception flows). A scenario has an id, a description, a *from step clause* (a list of references for existing steps), a list of steps, and a *to step clause* (also a list of references for existing steps). A step has an id, a a specification in the form of a tuple (user-action x system-state x sytem-response), and a list of annotations that can be used to semantically identify the step (avoiding the ~~problem of~~ fragile pointcuts). Finally, a reference to a step can be either a reference to a *step id* or to a *step annotation*.

which

is

talvez seja desnecessário

Listing 2: Use Case abstract syntax

```
data UseCaseModel = UseCaseModel Name UseCaseList
data UseCase = UseCase Id Name ScenarioList
data Scenario = Scenario Id FromStep StepList ToStep
data Step = Step Id Action State Response Annotations
```

The *Scenario Composition Weaver* is implemented by the *scWeaver* function (Listing 3), which takes as input the SPL use case model and a product specific list of scenarios (~~that may be~~ computed by the previous weaver). The SPL use case model is required in this composition because a product specific scenario is able to refer to a step that is defined in the SPL scenarios but was not selected in a specific configuration.

as

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

isso não deveria acontecer; não faz muito sentido

State em step? duas coisas diferentes

explica como estas funções lidam com ciclos

This implementation computes the complete paths of each product specific scenario by calling, recursively, the *completePaths* function. This function (lines 5-7 in Listing 3) takes as input the whole use case model (*ucm*) and a scenario (*scn*); and returns all complete paths (a list of *step lists*) of *scn*. The function *fromList* (called 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 a distributed list concatenation.

specifica

this is essentially a

After computing the complete paths, it is possible to derive a reasonable representation for the product specific scenarios. ~~We named this representation as~~ *trace model*, since it describe all possible sequences of events ~~produced~~ 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 process has occurred. The *traceModel* function (lines 9 and 10 of Listing 3) is responsible for computing this representation. ~~Such function~~ takes as argument another function *f* and a list of steps (*x:xs*). ~~Currently, the valid functions that can be applied as the first argument are~~ the *stepId* function (returns the id of a given step) and the *bind e* function (that also computes the parameter values of a given *step* in a specific *feature environment*). Therefore, if we are interested in the trace model representation, the *Scenario Composition Weaver* can be configured (the ~~*META*~~ element of our modeling framework) to return: a) a trace model without parameters, computed for each complete path of the resulting scenarios; or b) a trace model, with resolved parameters, for each complete path of the resulting scenarios. Next section we explain the *bind* and *feature environment* constructions.

onde você explica isso?

The model of this weaver is depicted in Table 5.

confuso

Table 5: Model of Scenario Composition Weaver

Element	Description
o	List of concrete scenarios or trace models
o_{jp}	Scenarios and steps of scenarios
L	{use cases and scenarios}
L_{ID}	From step and to step clauses
L_{EFF}	Compose concrete scenarios
L_{MOD}	Use cases and scenarios
meta	Function used to parameterize the trace model generation

não seria complete paths?

The output (o element of our modeling framework) can be either the concrete product specific scenarios, computed directly from *scWeaver* function; or a trace model, computed by calling the *traceModel* function for each concrete scenario. The input languages (L) are both the SPL use case model and the product specific scenarios. These (similar) input languages correspond to the parameters of *scWeaver* function.

the

não está legal, falar comigo

este texto dá a entender que são dois weavers!

Listing 3: The *completePaths* and *traceModel weaver* functions

```
scWeaver :: UseCaseModel -> ScenarioList -> [StepList]
scWeaver ucm [] = []
scWeaver ucm (x:xs) = (completePaths ucm x) ++ (scWeaver ucm xs)

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

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

?

pode omitir isso, assume que extractParams faz isso

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, should 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. Finally, as we have explained before, such mechanism is parameterized (the *META* element of our customized framework) in order to select, once the trace model was selected as the output language, if the parameters will be resolved or not.

dic

explica melhor isso

minúsculo

3.2.3 Bind parameters weaver

This weaver is responsible for the third activity of our variability management approach. 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 5 depicts the *Buy Products* scenario that can be reused for different *ship methods*. Without this parameterized specification, and aiming at automatically generating, for example, 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 in 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 are proposing an environment that relates them. Features related to parameters must be either an **alternative feature** or an **or feature** [18, 12, 11].

amarrações desnecessárias / melhor separar conceitos

to

of

The implementation of this weaver consists in a call to the *traceModel* function (Listing 3) with the *bind e* partial function as first parameter. The *bind* function (lines 1-5 of Listing 4) takes as input an environment (e), that maps a parameter into a feature, and a step (s). Then, it extracts all parameters from s, and returns a suitable event representation with the corresponding parameter values. Each text between the symbols "<" and ">" (defined in the user action, system state, or system response of a step) is treated as a parameter and must be defined in the environment (otherwise, an type exception is thrown).

Só uma?

For example, if a product is configured with either *Fast* and *Economical* ship methods, the result of applying this weaver for the Step 4M of the Buy Product basic version will result in the following representation: 4M(Fast, Economical)

deveria estar em L

The *extractParameterValues* function (called at line 5 of Listing 4) is responsible for extracting the related parameter values from a feature. Also, each parameter must be related to an **alternative feature** or to an **or feature** present in a product configuration.

desse

mapeamento é derivado do Product Configuration

Listing 4: The *bind weaver* function

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

Não existe este acoplamento

Table 6 describes

~~The description of the Bind Parameters model is showed in Table 6.~~ This weaver just resolves parameters in the trace model representation. Therefore, its output language is also the set of all valid sequences of events (a trace model) with binding parameters (join points). The use case model (UCM) defines the list of scenarios that might be parameterized (UCM_{EFF}). Each scenario parameter (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. The environment (e parameter of the *bind* function) is used for relating parameters to features.

melhor separar concerns

Table 6: Model of Bind Parameters Weaver

Element	Description
o	Trace model with resolved parameters
o_{jp}	Each resolved parameter
L	{UCM, CK}
UCM_{ID}	Parameter in the use case model
CK_{ID}	Optional and alternative features
UCM_{EFF}	Declares parameterized scenarios
CK_{EFF}	Defines the domain value of parameters
UCM_{MOD}	Parameter in the use case model
CK_{MOD}	Selected features
meta	

?

acho que está faltando o fundo, fala comigo

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)**, which was proposed for learning about and experimenting with software product line and that is applied for 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 devices.

Based on the last application of our approach, we reported about the benefits of a clear separation between variability management and scenario specification [6]. In this referred work, we compared our approach with PLUC and PLUSS techniques. This comparison was done based on Design Structure Matrices (DSMs), on a suite of metrics for quantifying modularity and complexity of specifications, and on observations of the effort required to introduce SPL increments (such as new variants or products).

For now, let us reproduce the DSMs to understanding the benefits of SoC in variability management. DSMs is an interesting and simple tool for visualizing dependences between design decisions [4]. Such decisions are distributed in both rows and columns of a matrix. We can identify which input data is required (a dependency) by observing which columns are marked in its corresponding row [4]. As presented in Section 2, 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 12, 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 12: DSM Analysis of PLUC

Our approach reduces the dependences between variability management and scenario specifications (Figure 13). 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, 13], and might be required in other artifacts too. Notice that the environment, 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.

In the remaining of this section we present the results of applying the metric suite for the *MMS* and *Pedagogical* product lines. We compared our specification of MMS product line to the specifications written by us in both PLUC and PLUSS techniques. All specifications are available at [2].

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

Figure 13: DSM Analysis of the proposed approach

In a similar way, we compared our specification of pedagogical product line to a specification that was sent to us by the authors of PLUSS approach.

The metric suite, used in what follows, was adapted from [17] and 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. Instead, *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 be interested 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, three metrics related to complexity were applied in this work. The first one, *Vocabulary Size*, quantifies the number of use cases (VSU) and scenarios (VSS) required by each of the evaluated approaches. The second one, *Steps of Specification* (SS), is related to the size of each scenario and identifies how many pairs *User action* x *System response* compose a specific scenario. Additionally, we also relate modularity to complexity by applying *Features and Steps of Specification* (FSS), which counts the number of steps of specification whose main purpose is to describe the behavior of a feature. A complete description of these metrics can be found elsewhere [6]. Next we present the quantitative analysis of MMS and Pedagogical product lines, which consider the metric suite just introduced.

4.1 MMS Product Line

As explained earlier, the MMS product line, which is a real case study, 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) [6]. We have specified the MMS product line using three techniques: our notation, PLUC, and PLUSS approaches. After, 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 14, by applying our approach we can reduce the diffusion over scenario of such features.

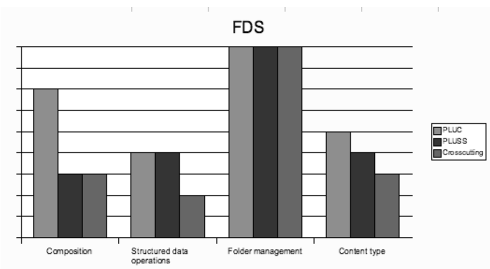

Figure 14: Relative FDS for evaluated techniques

4.2 Pedagogical Product Line

We compared our approach against two specifications of the Pedagogical Product Line(PPL): the original one, 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 several use cases (see modularity metrics in Table 8). Moreover, another characteristic of PPL is that some features are related to qualities that do not cause effect into use cases.

Table 8: PPL quantitative evaluation

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

Still in this context, our approach also achieve some improvements in the resulting specifications. The main factor for these improvements in the *Pedagogical* product line was the error handle modularization. Both in SEI and PLUSS specifications, the *error handler* concern was spread in several use cases.

However, by applying our approach, all behavior related to the *error handler* concern was specified in a single use case. The composition of *error handler* with the basic scenarios was done by means of annotations attributed to the corresponding steps. For instance, Figure 15 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 15: 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 needs to save information in the file system. Therefore, both in the SEI and PLUSS specifications of *Pedagogical* product line, several use cases were specified with scenarios for handling this kind of exception.

As a consequence, we achieve a significant 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 means an improvement in the specification reuse.

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 [28]. 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 [22]. 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 [27, 10]. 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 [8]. 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 argued about the crosscutting nature of variability management. Since a variant feature might require variant points to be spread in many artifacts, it does not make sense to tangle variability management with the other product line artifacts.

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

As future work, we want to identify the relationship between variabilities at different SPL artifacts (in a first moment, test artifacts). Such kind of relationships can help in the product generation and traceability activities of a SPL. The modeling framework proposed in this work takes a step in this direction, since the composition process used to derive product specific scenarios have been already implemented.

7. REFERENCES

[1] Ample project, online: http://ample.holos.pt/ (2007).

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

melhor o link direto

[3] V. Alves, A. C. Neto, S. Soares, G. Santos, F. Calheiros, V. Nepomuceno, D. Pires, J. Leal, and P. Borba. From conditional compilation to aspects: A case study in software product lines migration. In *Fifth International Conference on Generative Programming and Component Engineerig (GPCE'06)*, Portland, USA, 2006.

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

[5] 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.

[6] 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.

[7] G. Cabral and A. Sampaio. Formal specification generation from requirement documents. In *Brazilian Symposium on Formal Methods, SBMF'06*, Natal, Brazil, sep 2006.

[8] 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.

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

[10] 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.

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

[12] 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.

[13] M. de Medeiros Ribeiro, P. M. Jr., P. Borba, and I. Cardim. On the modularity of aspect-oriented and other techniques for implementing product lines variabilities. In *First LA-WASP, affiliated with SBES'07*, João Pessoa, Brazil, oct 2007.

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

[15] 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.

[16] C. Feitosa and A. Mota. Unifying csp models of test cases and requirements. In *Brazilian Symposium on Formal Methods, SBMF'06*, Natal, Brazil, sep 2006.

[17] 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.

[18] 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.

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

[20] 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.

[21] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

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

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

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

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

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

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

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

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

[30] B. Roscoe. *The theory and practice of concurrency*. Prentice-Hall, 1998.