Towards Consistency between Features and SPL Use Cases

Mauricio Alférez
Centro de Investigação em Informática e Tecnologias da Informação - CITI
Universidade Nova de Lisboa
Caparica, Portugal
mauricio.alferez@di.fct.unl.pt

Roberto E. Lopez-Herrejón
Institute for Systems Engineering and Automation
Johannes Kepler University
Linz, Austria
roberto.lopez@jku.at

Ana Moreira
Centro de Investigação em Informática e Tecnologias da Informação – CITI
Universidade Nova de Lisboa, Caparica, Portugal
amm@di.fct.unl.pt

Vasco Amaral
Centro de Investigação em Informática e Tecnologias da Informação - CITI
Universidade Nova de Lisboa
Caparica, Portugal
vasco.amaral@di.fct.unl.pt

Alexander Egyed
Institute for Systems Engineering and Automation
Johannes Kepler University
Linz, Austria
alexander.egyed@jku.at

ABSTRACT
In variability modeling for Software Product Lines (SPL) ensuring consistency between feature models in the problem space and their realizations in the solution space, is crucial to compose correct products. Currently, consistency checking — the description and verification of semantic relationships amongst the models — has focused on code-based modules and is not used for requirements analysis techniques. The objective of this paper is to establish a set of rules and constraints to support the consistency between features and use case models in the SPL domain, using feature models and a requirements-specific composition language based on aspect-oriented principles. This work aims at composing good quality models that help to understand the intended target products and are adequate for its integration in model-driven development processes.

Categories and Subject Descriptors
D.2.1 [Software Engineering]: Requirements/Specifications
D.2.13 [Software Engineering]: Reusable software – domain engineering.
D.2.2 [Software Engineering]: Design tools and techniques.
F.3.1 [Theory of Computation]: Specifying, verifying and reasoning about programs.

General Terms
Algorithms, Documentation, Design, Reliability, Languages, Theory, Verification.

Keywords
Software Product Lines requirements, use case models composition, consistency checking.

1. INTRODUCTION
Software Product Line (SPL) engineering refers to the techniques for the creation and management of products’ families for a particular domain using a shared set of assets [2]. In SPLs, features are useful to express product functionalities concisely [3]. There are common features between all the products in the product line (also called mandatory features or commonalities), and there are variable features that allow distinguishing between products in a product line (also called variabilities). To express variability, variability modeling is used to describe the different features available in an SPL and their interdependencies [4].

A wide used representation to model variability is the feature models, which belong to the problem space and their realization (i.e., their correspondent model- and code-based artifacts) is part of the solution space [4]. To produce particular products from an SPL, feature realizations in the solution space have to be composed according to a specific selection of features from a feature model usually called feature model configuration. This process requires a mapping between features from a feature model, and the models that realize them.

Traditionally, SPL engineering has been mainly focused on the realization of features and their mapping with code-base artifacts. However, a software artifact can be any kind of information that is part of or related to software, such as models and textual documentation [3, 5]. In particular, use case models in combination with feature models in SPL development enable stakeholders to derive the requirements of target software products and, more importantly, to reason about them [3, 6, 7]. In this paper we focus on model-driven development of SPLs and use cases [8] as a requirements specification technique.

A number of different approaches have been proposed to allow creating mappings between features from feature models and solution-space models [9-11]. However, ensuring consistency between feature models and use cases models has not been addressed. In this context, by consistency we mean the conformance of given feature models, use cases models and their mapping, with constraints of their underlying metamodels. Note that consistency here goes beyond syntactical or semantic errors of each kind of model in isolation, for example, an actor that is not associated to any use case, a dangling node, a mapping of a feature with an already removed part of the use case, or mutually exclusive and
required couple of features. It means that we also take into account constraints that are not merely expressed in terms of only one language’s metamodel which are generally well supported by UML editors in the case of use cases models (e.g., using OCL) or feature model editors (e.g., using domain constraints expressing features interdependencies and hard-coded restrictions that constraints the construction of the models to conform to their meta-model). In particular, we are interested in verifying consistency using domain constraints (e.g., requires and excludes relationship in feature models), the meaning of the relationships between the elements inside use cases models (e.g., includes, generalization, and containment between the elements), and the way in which these elements in use cases are mapped with the features in the feature model.

To understand how consistency between features and use cases influence the correct composition of use cases, we employ a composition process for use cases that follows aspect-oriented techniques to modularize the definition of the composition of parts of the use cases models (also called model fragments) that realize features. In this process we use feature models and a requirements-specific composition specification language based on aspect-oriented principles such as VML4RE [11, 12]. In VML4RE some modules, called variants, play the role of aspects. Variants associate combinations of features to composition actions that play the role of advices and describe instructions that wrap usually complex underlying use case model transformations.

The objective of this paper is to establish a set of rules and constraints that support consistency between features and use cases, which facilitate the correct composition of use case models in the SPLs domain. For this we take into account domain constraints, the meaning of the relationships between the elements inside use cases models, and the way in which these elements are mapped with the features in the feature model and composed according to VML4RE specifications. Previous work [1] addressed consistency in composition in multi-view modeling in SPL following a FOSD [13] approach to model composition for class, state machine, and sequence diagrams. This paper extends this work to requirements specifications using use case models and also for an AOSD approach to model composition within the VML4RE framework.

The remainder of this paper is structured as follows. Section 2 describes the models used for use case models composition and motivates consistency of features and use cases. Section 3 presents some rules and constraints to support consistency and Section 4 discusses its importance in Model-driven Development. Section 5 presents related work in consistency of features and use cases models, and Section 6 outlines future work and concludes the paper.

2. BACKGROUND AND MOTIVATION

To understand consistency between features and use cases models it is necessary to explain the use of some models (e.g., feature models, use case models, a mapping model between features and use cases) and a composition specification language.

2.1 Models Used in Composition

Feature Model and Product Configurations. Semantically, a feature model describes a set of all possible valid configurations of systems (also called products) [14]. A configuration specifies a concrete product in terms of its features. Figure 1-Left shows a sample feature model of part of the Smart Home SPL [15, 16]. Smart Home has two optional features, Electronic Windows (EW) and Smart Heating (SH). Also, it has a set of common features, such as Manual Windows (MW) and Manual Heating (MH) that will be included in all the target products to be produced using the Smart Home SPL.

The remaining of Figure 1 shows two sample product configurations of the Smart Home SPL that will be used to illustrate the consistency problems, Product-1 and Product-2 which are differentiated by the optional features that each one contains.

Use Case models. Feature models are complemented typically with other models such as use case models. In use case requirements modeling, model fragments are specific sets of elements that can be modeled in a use case model, such as use cases, packages, actors, their intra relationships and the relations with other fragments in the use case models. Use case models describe a high-level view of the intended functionalities of a SPL, which can vary according to the set of features chosen for a specific target product.

Figure 2-up shows part of the final target model built for Product-1. In Figure 2 it is important to understand the meaning of the relationships between the use case models’ fragments, especially, the include relationship. This is a relationship in which one use case (the base use case) includes the functionality of another use case (the inclusion use case). The include relationship supports the reuse of functionality in a use case model and is used to express that the behaviour of the inclusion use case is common to two or more use cases [17]. Note that include relationships between use cases may imply to constrain the relationship between the features related to them. For example, the include relationship between the base use case CtrlTempAuto that includes the inclusion use case OpenAndCloseWindow may imply that feature Smart Heating (SH) requires of the feature Electronic Windows (EW). We discuss this and other consistency constraints in Section 3.

Mapping Model. Figure 2 shows use case model fragments, such as actors and use cases, which are related with the features shown in Figure 1. The base mechanism to relate requirements model fragments to features is to use a correspondence table (or mapping table), as presented by [18], [3] and [19]. Also, with the advent of MDD technologies, the use of tools such as Feature-Mapper [20], FMP [10, 21], and VML4RE [11, 12] cases the linking between features and other models such as base requirements models. The mechanisms used to link requirements models and features range from visual editors with facilities such as drag-and-drop and multiple selections of model fragments, to mechanisms to create links programmatically, for example, using quanti-
Composition Model. A composition model such as the VML4RE (Variability Modelling Language for Requirements), shown in Listing 1, guides the specification of the transformation of the models, such as the use case model for Product-1. VML4RE [11, 12] is a textual language that allows associating actions, that wrap a set of model transformations for specific requirements models such as use cases, to features expressions written as logic expressions. Feature expressions are atomic, or compound expressions. Atomic features expressions represent single features and compound feature expressions are atomic features associated with logic operators such as AND, NOT and OR. This works as follows: if Smart Heating (SH) is chosen to be part of a target product, for example, the feature expression (i.e., the terminal feature expression SH) in Listing 1 (line 4), will be evaluated to TRUE. The consequence of this is that the actions that are inside the SH variant block will be processed and applied to a base model (i.e., Listing 1, lines 5-9). For example, the CntrlTempAuto will be inserted into the package Heating (Listing 1, line 5) and then it will be related to other use cases using includes and extends relationships (Listing 1, lines 7-8). If more than one feature expression is evaluated to TRUE, the default composition order follows a top-down sequence.

2.2 Use Case Models Composition

Figure 2 shows Product-1 and Product-2, built using the composition model shown in Listing 1. We annotated the models with numbers that represent the line in Listing 1 where an action is specified. Note that we omitted some of the actions, for example, the insertion of some actors such as WinSensor and WinActuator.

Figure 2-up shows Product-1 and corresponds to the feature model configuration in Figure 1-middle where EW and SH are selected. Because both feature expressions (EW and SH) evaluated to TRUE, the default composition order follows a top-down sequence and Product-1 is built applying first the variant EW and then variant SH. This is an example where features and use cases are consistent because allow the composition of all the model fragments that realize the intended functionality of the selected features. It means that all the functionality represented in the use case model that realize the features selected in the feature model configuration, are in fact included in the correspondent target product.

Listing 1. Composition variants for Smart Heating and ElectronicWindows

```
1  | variant for EW {  
2     | insert useCase OpenAndCloseWinAuto; //...more actions  
3     }  
4  | variant for SH {  
5     | insert useCase CntrlTempAuto into package Heating;  
6     | insert useCase CalcEnergyConsumption into package Heating;  
7     | insert includes from CntrlTempAuto to OpenAndCloseWinAuto, AdjustHeaterValue;  
8     | insert extends from CntrlTempAuto to CalcEnergyConsumption;  
9     }  
10  }  
```
that represent functionality that are required in the product according to the semantic of the use case model. That happens in Product-2 where, despite the fact that it satisfies the domain constraints imposed by the feature model, it does not compose a use case model that represents the functionality that is expected to have in a product built in the SPL. In this case, OpenAndCloseWinAuto functionality that is not present in the model but is required through the “include” relationship with the CtrlTempAuto use case.

For small examples with a reduced number of features, it is easy to control consistency problems such as the one mentioned previously. One solution for our example would be to guarantee the presence of the feature SH when feature SH is selected, in every possible feature model configuration. Other solution is to establish that SH will be a mandatory feature in the SPL. This will allow the execution of the actions inside variant SH that place the missing model fragments (e.g., the use case OpenAndCloseWinAuto) in the model. However, the number of possible combinations of features in a feature model may grow exponentially with the number of features. The result of this explosion is that we cannot verify all them manually to see if these are consistent with the use case model of the products. Next section shows some rules and constraints that help to detect specific instances of these inconsistencies.

### 3. DOMAIN AND COMPOSITION CONSTRAINTS

According to Czarnecki et al [22] and Lopez-Herrejón el al [1] composition constraints should follow from domain constraints. Let PLf denote the domain constraints that can be derived from a feature model of an SPL, and Cf denote composition constraints that will be derived in Section 3.2. We use propositional logic to express and relate PLf and Cf. Because we are interested in verifying that all members of the product line satisfy a given composition constraint, the following formula should not be “satisfiable” [1]:

\[
\neg (PL_f \Rightarrow C_f)
\]

If it is satisfied, it means that there is a member of the product line that does not meet constraint Cf. By using a satisfiability (SAT) solver, the violating feature configuration(s) can be identified. This is done for each instance and each composition constraint we want to verify.

#### 3.1 Deriving Domain Constraints (PLf)

Figure 3 summarizes how PLf can be obtained by mapping feature models to propositional formulas based the research work of Batory [23] and Benavides [24]. Considering the feature model shown in Figure 1 (left) and the mapping to propositional logic in Figure 3, the PLf for the Smart Home SPL example is presented in equation 2.

\[
\begin{align*}
\neg (\text{SmartHome-SPL} \Rightarrow \text{true}) \land \\
(\text{SmartHome-SPL} \Rightarrow \text{WM}) \land (\text{SmartHome-SPL} \Rightarrow \text{NM}) \land \\
(\text{WM} \Rightarrow \text{WM}) \land (\text{EW} \Rightarrow \text{WM}) \land \\
(\text{HM} \Rightarrow \text{HM}) \land (\text{SH} \Rightarrow \text{HM})
\end{align*}
\]

Next section identifies an initial set of constraints that are essential to use case composition (Cf).

#### 3.2 Deriving Composition Constraints (Cf)

Composition constraints act as consistency rules describing the semantic relationships that must hold amongst the different models fragments. First we obtain the constraint equation expressed as a logical expression and then we relate it to the rules.

##### 3.2.1 Creating the Constraint Expression

Let F be a feature that refers to a model element e defined in another feature. To be consistent, a SPL product that includes feature F must also include at least one other feature Freqi where element e is defined. This is denoted in the following equation [1]:

\[
C_f \equiv F \Rightarrow \bigvee_{i=1}^{k} F_{reqi}
\]

By substituting Cf of Equation 3 in Equation 1, we obtain the logical expression in Equation 4, which can be passed to a SAT solver [1]:

\[
\begin{align*}
\neg (PL_f \Rightarrow C_f) & \equiv \neg (PL_f \Rightarrow (F \Rightarrow \bigvee_{i=1}^{k} F_{reqi})) \\
& \equiv \neg (\neg PL_f \lor \bigvee_{i=1}^{k} F_{reqi}) \\
& \equiv PL_f \land F \land \neg F_{reqi}
\end{align*}
\]

Therefore expression 4 evaluates to false when feature F requires an element or set of elements in the use case model that were not introduced in the use case model because none of the correspondent features Freqi that introduce these elements were selected to be part in the product.

##### 3.2.2 Defining the Rules

**Rule 1 - Required Inclusion Use Case:** it specifies that at least one feature (Freq) defines an inclusion use case inclusion and it is selected in every feature configuration that contains the feature F which introduces a base use case related to inclusion.

For example, given that SH is selected (i.e., F=SH), and it is related to the base use case CtrlTempAuto, we want to guarantee...
that there are at least one feature (e.g., $Freq_i=HM$ in equation 4) linked to the inclusion use case AdjustHeaterValue and at least one feature (e.g., $Freq_i=EW$) is linked to the inclusion use case OpenAndCloseWinAuto, and that $HM$ and $EW$ are selected in the feature model configuration. This way, we guarantee the presence of the functionality required by CtrlTempAuto, such as regulate the temperature by opening and closing windows and adjusting the internal heater value.

Rule 2- Required Package: it specifies that at least one feature ($Freq_i$) that defines a package package must appear in the SPL product that contains a feature $F$ that defines elements such as use cases, associations, includes or extends relationships inside package.

For example, given that the use case CalculateEnergyConsumption is inside the package Heating and they are related to $SH$ and $HM$ respectively, we have to guarantee that product configurations that contain $SH$ (i.e., $P=SH$) also contain the feature $HM$ (i.e., $Freq_i=HM$). That way we maintain the system modularization and decomposition based on packages. To maintain the base modularization of the system also contributes to avoid that composition actions reference to undefined (i.e., not inserted) elements inside packages. For example, the insertion of use cases inside the Heating package as shown in Listing 1 (line 1), “insert useCase CtrlTempAuto into package Heating”. One of the mayor problems with references to undefined elements is that it prevents the application of composition actions that are intended to build a correct and complete model of the target product based on a specific selection of features.

In the feature model of Figure 1-left we already determined that $HM$ is a mandatory feature, therefore $HM$ will be present in all the feature model configurations. Otherwise, we would have to create a relationship of requires from $SH$ to $HM$ to guarantee that always that $SH$ is selected in a configuration, $HM$ will be also selected.

Rule 3- Required Generalization: it specifies that at least one feature ($Freq_i$) that defines an element $elementA$ that generalizes other element $elementB$, must appear in the product that contains the feature $F$ that introduces $elementB$.

In use cases models, generalization applies to actors and use cases. Figure 2 shows an example of generalization of use cases. Given that the use case OpenAndCloseWindows is a generalization of OpenAndCloseWinAuto and they are related to $WM$ and $EW$ respectively, we have to guarantee that $WM$ is selected for every product that includes $EW$. This is necessary because the child model elements (also called the “specialization”) in generalizations inherit the attributes, operations, and relationships of the parent (also called the “generalization”). Therefore developers must only define for the child the attributes, operations, or relationships that are distinct from the parent. A consequence of this is that the only composition of the child excluding the parent will result in missing attributes, operations, or relationships that are needed by the child and to build a complete target model.

In the feature model of Figure 1-left we already determined that $WM$ is a mandatory feature, therefore $WM$ will be present in all the feature model configurations. Otherwise, we would have to create a relationship of requires from $EW$ to $WM$ to guarantee that always that $EW$ is selected in a configuration, $WM$ will be also selected.

4. DISCUSSION
A difficult task in the creation of model-based SPLs is to assure consistency between features and their realizations in models. An issue in this respect is the lack of properly described constraints related to the relationships between features and model fragments that affect the composition of the models. To the best of our knowledge, there are no catalogues of descriptions of these rules and constraints in requirements models such as use cases models. It is important for single systems and it is even more so when models are part of a SPL. Furthermore, when the models are used for MDD it becomes a crucial issue as software is built by means of a chain of transformations. This can start from assets such as requirements specification models, to code-based assets that typically depend on a particular implementation technology. In this setting, the quality of the final code of target products depends mostly on (i) the transformations, (ii) the source models of each transformation and (iii) the information added after each transformation. Therefore, to create constraints helps not only to produce composed models that helps to understand the intended target products to all the SPL stakeholders, but also to obtain good quality source models that are the base for deriving good quality code.

5. RELATED WORK
There are different research areas related to our work. For example, the field of well-formedness of models, for example Egyed [25]. Also, for single systems modeling, Jacobson [26] used use case models and employed aspect-oriented techniques. He uses use case slices to model separately each concern and then uses a merge strategy to compose all the slices to create a composed model. However, neither Jacobson nor Egyed verify consistency and their composition mechanism does not support model fragments weaving as it is possible with a requirements-tailored composition language as VML4RE. Closer to our application domain is the MATA approach [7]. It uses critical pair analysis provided by their underlying graph transformation tool AGG to detect what they call “feature interactions” in designs models. In a general sense, the model elements that participate in a feature interaction are similar to what we have called realizations of $F$ and $Freq_i$. We believe that mature algorithms and tools supporting graph-based transformations in AGG are an important way to understand consistency and composition, and there are synergies with our approach.

Also, in the domain of SPL engineering, previous work [1] addressed safe composition in multi-view modeling in SPL. This served us as a base to express our constraints and rules. However, it was focused on a specific FOSD approach [13] for class, state machine, and sequence diagrams. This paper extended this work to use case diagrams and also for AOSD modeling within the VML4RE framework.

6. CONCLUSIONS AND FUTURE WORK
This paper establishes a set of rules and constraints to support consistency of use case models in the SPL domain, using feature models and VML4RE. However, our approach does not depend on the use of VM4RE. We use it because its actions facilitate expressing the composition in use cases and encapsulate underlying complex model transformations. Also, VML4RE take advantage of the power of AOSD in because of the use of from two of its properties highlighted by Filman and Friedman [27]: quantification, which is the ability to declare changes to be applied consistently to many places of a program or model, and obliviousness,
that implies that the extended programs or models, do not need to be aware of being extended and do not need to provide special hooks for enabling extension.

The issues addressed here are not just about completeness issues or omissions as we show in the small motivating with dangling nodes or relationships. The point of consistency for composition is to guarantee that all product models that can be composed based on a feature model are consistent, meaning without omitting information that will be required in case the models that are processed and transformed and refined into more platform dependent models and code.

We think that the application of constraints as the ones presented here are necessary but do not satisfy completely the problem of correct composition. This is because a correct composition not only depends on the presence of features that introduce model fragments required by the model fragments introduced by other features. It also depends on the composition order of the variants and in the application order of the actions inside each variant block. In this moment, we are researching algorithms to calculate the precedence order between variants after the verification of the constraints included in this paper. Also, we are building a prototype tool that extends the one developed in [1] and we hope to evaluate its performance and scalability with Smart Home and some other case studies.

Other issues in composition are related to non-monotonic composition, i.e., composition that also includes changes and deletions of model fragments and inter-view rules that transcend the use cases model to cover other requirements or design models such as goal models and activity diagrams. These issues will be part of our future work.

ACKNOWLEDGMENTS

This work was partially supported by the European Project AMPLE, contract IST-33710, the grant SFRH/BD/46194/2008 of Fundação para a Ciência e a Tecnologia, Portugal. It was also partially funded by the Austrian FWF under agreement P21321-N15 and Marie Curie Actions—Intra-European Fellowship (IFE) project number 254965.

REFERENCES