Dealing with Constraints during a Feature Configuration Process in a Model-Driven Software Product Line

Hugo Arboleda[1,2], Rubby Casallas[1], Jean-Claude Royer[2]

[1]Software Construction Group, University of Los Andes, K 1 N° 18A 10, Bogotá, Colombia
[2]OBASCO Group (INRIA & LINA), Ecole des Mines de Nantes, La Chantrerie, 4 rue Alfred Kastler, 44307 Nantes, France
hugo.arboleda@emf.fr, rcasalla@uniandes.edu.co, jean-claude.royer@emn.fr

Abstract. We present our ongoing work on an approach to create Model-Driven Software Product Lines by means of successive model refinement, guided by configuration of features. Each refinement uses model-to-model transformation until arriving at the executable code with technological platform details included. During this process, users select features at each stage taking into account their preferences and requirements. The selection of features can be performed for each element of the model. Thus, the selection is constrained by many facts, for example, a mandatory selection element-feature because some structural model relationships that has to be preserved. To deal with model transformations while satisfying the constraints, we introduce the concept of constraint-model to restraint the possible feature configurations a user can specify. Then, we propose the construction of transformations by composing several rules that facilitate, from a single source, the generation of different targets according to a given feature configuration.

Key words: Software Product Lines, Product Line Architecture, Model Driven Architecture, Feature Model, Variability Management, and Model Transformation Rules

1 Introduction

Software Product Line Engineering (SPL) attempts to develop software products that satisfy specific needs for a segment of the market, by reusing sets of components called core assets. The core assets are developed during the domain engineering process, taking into account the variability and commonality of the product line, and are reused during the application engineering process [1].

Various approaches propose support for product lines creation. For instance, Compositional approaches use frameworks that are built using general-purpose languages and Generative Software Development approaches use textual or graphical Domain Specific Languages for generating automatically product line members. Furthermore, Model Driven Development (MDD) approaches suggest raising the abstraction level of product lines by using formal metamodels that describe Domain Specific Model Languages (DSMLs). MDD also suggests using high-level executable model transformations, and multi-stage strategies for derivation of product line members.

Feature modeling is a classical method and notation for capturing commonalities and variability of product lines. Feature modeling was originally introduced in [2], and since then an important number of extensions and variants have been proposed to improve its expressiveness [3][4][5].

In [5][6], product lines are created by using an MDD approach. In these, the authors suggest creating feature selections that define different products, which are constrained by feature model semantic or OCL sentences. This kind of approach can be used to manage that is called in [7] as negative variability. Managing negative variability means taking away parts of a general model (template) based on a feature selection.

In [8][9], we have presented an approach based on Model Driven Development (MDD) for creating SPLs. We do not only manage negative variability but we also manage positive variability. This means that we can transform models by adding new domain elements as opposed to taking away elements that already exist in the source models. In this approach, we separate domain concerns using design abstraction levels. Thus, for creating SPL members, we create one high-level design model ($level_{i=0}$) and we refine it successively (in many stages) by using model-to-model transformations until obtaining low-level design models ($level_{j>i}$). For each level, we create a base metamodel and a feature model for expressing the structural and non-structural variability respectively. Then, before performing a refinement, we allow the user to customize the model-transformation by selecting the set of features that he/she wants to obtain in the target model.

Thus we can raise the level of abstraction of product lines and narrow the gap between the problem and solution space through a generative strategy, and between the domain and application engineering using configuration languages as feature models. We also can delay low-level design decisions until the final stages of model refinements and broaden the product line scope by including multi-platform variation points.

However, we have identified the need of assisting the user in the feature configuration process by taking into account the requirements and the constraints during successive refinements. Specifically, there are constraints: in the way in which a user can relate model elements and features for customizing the model-transformations. The constraints can come from structural relationships, for instance, when the user needs to express the fact that one model element cannot be associated with a specific feature if another relationship already exists. To achieve this assistance, we introduce the concept of a constraint model that establishes, at the metamodel level, the valid feature configurations a user can make.

We have organized this article as follows. Section 2 describes the relation between features and transformation rules. Section 3 presents the constraint models and their properties. Section 4 explains the application engineering process using our approach and finally, Section 5 outlines our future work.

2 Features and Transformation Rules

In MDA, transformation rules are written in terms of metamodel concepts. Thus model elements are transformed using the transformation logic associated to the metamodel concepts that they conform to. To transform a model element into different targets, according to the features selected by the user, we create several transformation rules for each metamodel concept; each one depends on a possible feature a user can choose. Thus, during the refinement process, the user performs his/her selection not only by selecting features, but also by associating model elements to them. Each selection-association determines the transformation rule that will be used for transforming the associated element. The combination of those rules composes the whole transformation that will be applied at the corresponding stage of the refinement.

Fig. 1 introduces a small example based on a case study concerning a Smart Home Product Line (SHPL). Regarding some variability associated to a SHPL, the access control feature can be different if we talk about different doors or windows. Thus in the same house, one door can manage the access control using an user code and another door can manage it using the finger-print of the house's owner. The associations (dashed lines *a* and *b*) in the Fig. 1 imply that different transformation rules will be executed for transforming two elements (*MainEntrance* and *BackEntrance*) that conform to the same metamodel concept (*Door*). Fig. 2 illustrates how different transformation rules can be created for one metamodel concept and be related to different features.

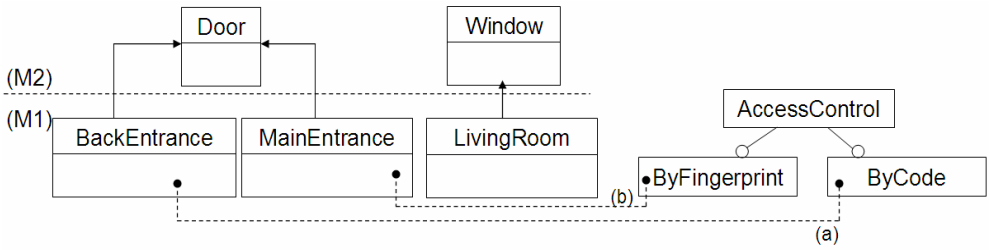

Fig. 1. The *MainEntrance* and *BackEntrance* are model elements that conform to the *Door* concept. The *ByCode* and *ByFingerprint* are features that the user can associate to model elements for creating an specific product. In this example, a user creates two associations. The first one is between *BackEntrance* and *ByCode* (*line a*), and the second one between *MainEntrance* and *ByFingerprint* (*line b*).

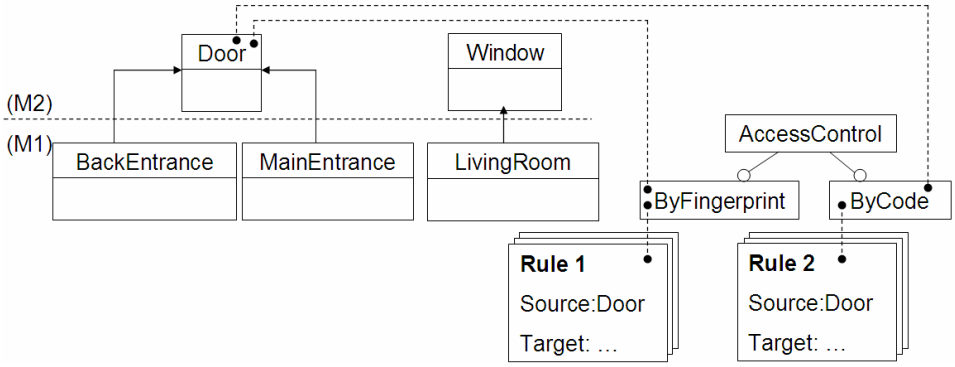

Fig. 2. The *Door* concept is related to two possible features. Thus, each feature is related to one transformation rule that will be executed if the user selects and relates one *Door* element and one of those features. For the example introduced in the Fig. 1, *Rule 1* will be executed for the *MainEntrance* element and *Rule 2* for the *BackEntrance* element.

3 Constraining the Feature Configurations

The model resulting from associating the metamodel concepts and the features (Fig. 2) is called the constraint model. A constraint model defines the possible allowed relationships between $level_i$ model elements and $level_{j=i+1}$ features. If there is an association between a metamodel concept and a feature, the user can create a relationship between any model element that conforms to the metamodel concept and the feature. Otherwise, the feature configuration is not valid.

For instance, in Fig. 2 a user cannot make a feature configuration between *LivingRoom* and *ByCode*, because there is not an association in the constraint model that relates the *Window* metaconcept to the *ByCode* feature. In terms of the smart home, it means that a window can not manage the access control using an access code. Thus, we manage the product line scope by constraining the possible options for creating smart home products. We define two types of properties for the associations created as part of a constraint model. These are the cardinality and the dependency constraint properties. Next sections explain these properties further.

3.1 Cardinality Property

The use of properties as part of the associations between metamodel concepts and features is motivated by the need of adding semantic to the constraint model. Let us assume that due to non-functional requirements, one and only one door should manage access control by fingerprint. Then, it is mandatory to define this information in the association between the *Door* concept, and the *ByFingerprint* feature. Fig. 3 illustrates this example by associating the semantic construction *[1..1]* to the relationship. This semantic construction is a cardinality property.

The cardinality property is a UML-like cardinality *[i..j]*, where 0<=*i*<=*j* and where *j* can be undefined (*). This property makes possible to restrict the number of feature configurations between model elements and feature nodes, *i.e.*, to express constraints such as "only one of this can have this feature". In our approach, the semantic of the cardinality property varies according to the type of feature (solitary, group or grouped) that is related to one metamodel concept.

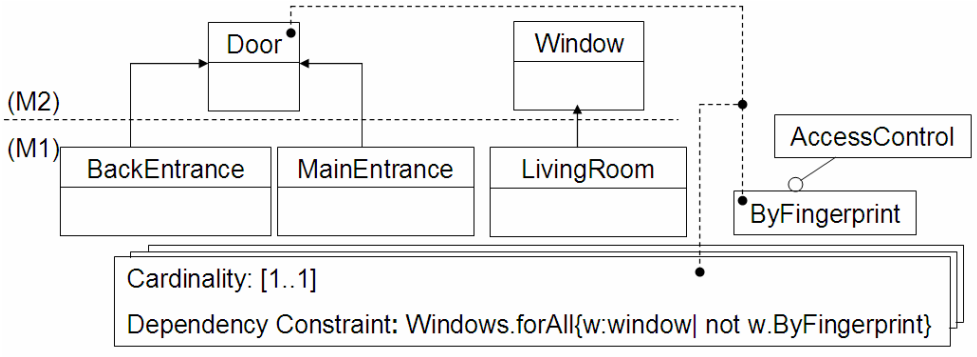

Fig. 3. Two properties, the *Cardinality* and *Dependency Constraint*, are associated to the relationship between the *Door* concept and the *ByFingerprint* feature for adding semantic to the constraint model.

3.2 Dependency Constraint Property

Suppose that due to non-functional requirements, a *Door* element can be related to the *ByFingerprint* feature, if and only if there is not a *Window* element related to the *ByFingerprint* feature. Then, for the cardinality property it is mandatory to define this information into the association between the *Door* concept and the *ByFingerprint* feature. Fig. 3 illustrates this example by associating to the relationship the sentence *Windows.forAll{w:window| not w.ByFingerprint}*, which is a dependency constraint property.

The dependency constraint property complements the cardinality property by taking into account the structural relationships of model elements and the already created relationships between model elements and features. This property is described as an OCL-type sentence that allows query models at different levels (M2 or M1). This kind of property allows expressing multiple types of constraints. More concrete examples of constraints and sentence constructions will be addressed in future work on more detailed examples.

4 Processes of Feature Configurations and Transformation

By having a constraint model between $level_i$ metamodel concepts and $level_{i+1}$ features, we can assist the user during the task of selecting features. Thus instead of validating feature configurations *a posteriori*, we propose to require him/her to create valid relationships. To achieve this assistance, we create a feature metamodel that extends the feature metamodel introduced by Czarnecki et Al. [5]. This makes it possible to clone features and associate them to typed attributes as *String* or *Integer*.

Continuing with our example, let us assume that we would like to create a relationship between the *MainEntrance* element in the $level_{i=0}$, and the *ByCode* feature in the $level_{j=i+1}$. Thus using the constraint model, the assisted process suggests either the *ByCode* feature or the *ByFingerprint* feature, if and only if, there is not yet a *Door* element related to the *ByFingerprint* feature (because of the cardinality property, see Fig. 3).

Since we create modular transformation rules associated to metamodel concepts, we need to compose them for creating valid model transformations by considering the possible feature configurations. In [8][9], we suggested to create three types of transformation rules. These types of rules

are: (1) base rules, (2) control rules, and (3) specific rules. Base rules are used to create the commonalities of product line members. Control rules are used to evaluate feature configurations and trigger specific rules, which are used to create the variability of product line members. According to this strategy, we create complete model transformation programs that are composed of many transformations rules.

5 Discussion and future work

In this article, we presented an approach to express constraints in the association between model elements and features. Currently, we are defining the refinement process to transform the original model into the source code. At each refinement, we need to take into account the selection of features done by the user but still satisfying the constraints.

For the creation of constraint-models, we are designing a metamodel based on the feature metamodel introduced by Czarnecki et Al. [5] for allowing relate metamodel concepts to features adding semantic represented by the properties introduced in this paper. Specific challenges of this element of the proposed solution are related with the creation of an Eclipse plug in, the extension of existent plug-ins as the presented by Antkiewicz et Al. [10], the use of model weaver tools as AMW [11], or the reuse and extension of oAW and XWeave [12].

6 Acknowledgments

This work is partially supported by AMPLE Grant IST-033710 and COLCIENCIAS – Colombian Institute for Development of Science and Technology "Francisco José de Caldas".

References

1. P. Clements, L. Northrop. Software Product Lines – Practices and Patterns, Addison-Wesley, 2002.
2. Kang, K., Cohen, S., Hess, J., Nowak, W. and Peterson, S. Feature-oriented domain analysis (FODA) feasibility study, Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA, 1990.
3. Lee, K., Kang, K. C. and Lee, J. Concepts and guidelines of feature modeling for product line software engineering. Proceedings of the 7th International Conference on Software Reuse. Apr. 15-19, 2002, Austin, USA. Vol. 2319 of Lecture Notes in Computer Science, Springer-Verlag, Heidelberg, Germany, pp. 62–77.
4. Czarnecki, K., Helsen, S. and Eisenecker, U. Staged configuration using feature models, in R. L. Nord (ed.), Software Product Lines: Third International Conference, Boston, USA, Aug. 30 - Sept. 2, 2004. Proceedings, Vol. 3154 of Lecture Notes in Computer Science, Springer-Verlag, Heidelberg, Germany, pp. 266–283.
5. Czarnecki, K., Kim, C. H. P. Cardinality-Based Feature Modeling and Constraints: A Progress Report. In OOPSLA International Workshop on Software Factories. San Diego, USA, Oct 2005. Paper available at http://www.ece.uwaterloo.ca/~kczarnec/sf05.pdf.
6. Czarnecki, K., Antkiewicz, M. Mapping features to models: A template approach based on superimposed variants. In proceedings of GPCE 2005. Volume 3676 of Lecture Notes in Computer Science, pp. 422-437.
7. Voelter, M., Groher, I. Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. Proceedings of the 11th Software Product Line Conference. Sept 10 – 12, 2007, Kyoto, Japan.
8. Garces, K., Parra, C., Arboleda, H., Yie, A., Casallas, R.: Variability Management in a Model-Driven Software Product Line. Avances en Sistemas e Informática, Vol. 4 No. 2, Sept 2007. pp. 3–12.

9. Arboleda, H., Cassallas, R., Royer, J-C. Implementing an MDA Approach for Managing Variability in Product Line Construction Using the GMF and GME Frameworks. 5th Nordic Workshop on Model Driven Software Engineering. August 27 – 29, 2007. Ronneby, Sweden. pp. 67–82.
10. Antkiewicz, M., Czarnecki, K. Feature Plugin: Feature modeling plug-in for Eclipse, OOPSLA Eclipse Workshop on Technology eXchange. Oct 24, 2004. Vancouver, Canada. pp. 67 – 72.
11. Didonet Del Fabro, M., Jouault, F.: Model Transformation and Weaving in the AMMA Platform. In Proceedings of the Generative and Transformational Techniques in Software Engineering Workshop. Jul 2005. Braga, Portugal. pp 71-77.
12. oAW, openArchitectureWare site. On line: http://www.openarchitectureware.org/.