Managing variability in multi-views engineering: A live demo

Marie Gouyette, Olivier Barais, Jérôme Le Noir, Jean-Marc Jézéquel

To cite this version:

Managing variability in multi-views engineering

A live demo

Marie Gouyette * — Olivier Barais ** — Jérôme Le Noir *** — Jean-Marc Jézéquel **

* INRIA
Campus universitaire de Beaulieu
35042 Rennes cedex, France
Marie.Gouyette@inria.fr
** IRISA, Université Rennes 1
Campus universitaire de Beaulieu
35042 Rennes cedex, France
{barais, jezequel}@irisa.fr
*** Thales Research and Technology
Campus de Polytechnique
1 avenue Augustin Fresnel
91767 Palaiseau Cedex, France
jerome.lenoir@thalesgroup.com

RESUMÉ. Cet article présente une suite d’outils permettant la specification de la variabilité dans les langages de modélisation dédiés à une précoccupation d’un système. Il est composé d’un outil graphique permettant de modéliser la variabilité d’une ligne de produit à l’aide de feature models référençant des éléments des modèles métier. Un autre outil permet de dériver un produit à partir de la ligne de produits définie (Product Derivation). Enfin un troisième outil fournit un moyen de visualiser les informations de variabilité dans les éditeurs de domaine.

Cette suite d’outils a été développée et testée dans le cadre du projet ANR Movida qui traite de l’ingénierie multi-vue dans le cadre de l’ingénierie dirigée par les modèles(IDM).

ABSTRACT. This paper presents a tool-suite to model variability of a Software Product Line with feature models in the context of multi-views engineering. This tool-suite proposes four modules: (i) to model variability of views using a feature diagram, (ii) to select features required for a specific product, (iii) to derive a product and (iv) to visualize variability information directly inside the base-model editor. This tool is developed and tested on the ANR Movida project which deals with multi-views engineering as part of Model-Driven Engineering.

MOTS-CLÉS : Ingénierie multi-vues, Lignes de Produits, Ingénierie Dirigée par les modèles, Outils

KEYWORDS: Multi-views engineering, Software Product Lines, Model-Driven Engineering, Tools

1. Introduction

Today, and likely for a long time to come, the complexity of software dominant systems is still growing and the variety of system classes tends to expand. From embedded systems which are required to cope with spare resources, to system of systems for which the evolvability and flexibility is key, requirements classes are expanding. In addition new concerns or more stringent existing concerns bring their extra complexity. They are environmental concerns, maintenance, repair and operation (MRO) concerns, supply management concerns etc. All of them play today an active or even sometime decisive role in the engineering decision process. The difficulty to embrace the whole complexity of the concerns and the difficulty to manage their inter-relations has raised the interest of the engineering community for “concerns driven” engineering. This is addressed today in the model driven engineering research community through the exploration of “viewpoint modelling” technologies. Viewpoint engineering permits to separate different concerns of the same system (such as Safety or Exchange for example) model in DSML (Domain Specific Modelling Language). So in viewpoint-based approaches (cf Sommerville and al. [SOM 97]), these different concerns are represented in views which focus only of a given concern.

In the MOVIDA (MOdelling Vviews and Decision support for Architects) project, a French ANR project that deals with viewpoint engineering in Model-Driven Engineering (MDE) context, we address the modelling of variability in views designed with DSML. Indeed, to simplify the creation of products and reduce cost, many companies such as Thales propose variation of already existing products. These products have commonalities and some variations between them. We can regroup these products into families. For software systems, the Software Product Line (SPL) community [POH 05] offers many techniques and tools to manage a family of products. A Designer can retrieve on SPL not only the specificity (variability) of a given product but also commonalities between the different products. Another kind of SPL is Model-Driven SPL [PER 08]. In this case, product line and derived products are software models. The MOVIDA project [INR 10] proposes to create and use a family of views : models with the same concern designed with domain specific modelling languages using a Model-Driven SPL.

The goal of this paper is to discuss new problems that should be managed when we want to combine viewpoint engineering and variability modelling. This paper also presents a tool suite to model variability across views designed with DSML.

This paper first presents the background and the related work used for building this tool suite. In section 4, we present the feature diagram tool suite, its architecture and the technologies used to create the feature diagram editor. Finally we conclude in presenting future work.

2. Background

2.1. MOVIDA Overview

MOVIDA is a French ANR project for 3 years (2009-2011). The aim of the MO-
VIDA project is to provide a support to model-driven viewpoint engineering through:

- The definition of viewpoints (a given concern)
- The detection of inconsistencies (for example between data from different viewpoints)
- The definition of representations for these viewpoints
- The modelling of composition (for team working on different models according to different views)
- The modelling of variability (common and variation point between different views)
- The multi-criteria analysis to select the better compromise of architecture and support architect decision.

Five partners take part of this project, two academics (UPMC (Université Pierre et Marie Curie) and INRIA Rennes-Bretagne Atlantique), and one industrials (Thales) and one SME (Obeo).

2.2. Viewpoint-based approach and Variability

The viewpoint-based approaches [SOM 97, CAR 03] permit the separation of different concerns in a system (like for example performance or safety viewpoint). So, each concern (viewpoint) is represented by a specific domain metamodel. However, the different views of a family of system can share commonalities and propose some specificities like in Software Product Line (SPL). Consequently, a viewpoint-based approach must support variability modelling. Variability can be model by many ways, one of them is using the concept of feature modelling. Eisenecker et al. [ULR 00] define features as “a distinguishable characteristic of a concept (e.g system components and so on) that is relevant to some stakeholder of the concept”. These features are linked each other thanks to operators which permit to express possible choices between feature children of a given feature. In the Movida’s context, the variability is used to manage commonalities and specificities of different view models. This variability uses the same mechanisms as the Model Driven SPL: Feature model to capture variability bound to domain model elements. Next section presents related work on feature diagram approaches.

3. Related work

3.1. Feature model notations

Variability can be expressed using a feature diagram approach defined by Kang et al. [KAN 90]. This is the first feature diagram approach. Other approaches exist such as FORM. FORM (Feature-Oriented Reuse Method) has been proposed by Kang et al. [KAN 98] as an extension of FODA. The new features diagram can not only trees
but also Directed Acyclic Graph (DAGs). Moreover, two new kind of relationship between features where added, generalization/specialization of features and implemented by. Features are separated between layers according to their use. So four layers are available, capability layer, operating environment layer, domain technology layer and implementation technique layer. There are few changes in term of graphical notation, except that the features are now in boxes. There are also many other notations such as FeatuRSEB (Griss et al. [GRI 98]), van Gurp et al. ( [GUR 01]), Generative Programming (Czarnecki and Eisenecker [ULR 00], Riebisch et al. ( [RIE 02] which use only operator of multiplicity, cardinality, and notion of optional and mandatory ) or PLUSS(base on FeatuRSEB [ERI 05]).

3.2. Mapping between features and Domain Model Elements

Heidenreich et al. [HEI 09] present a study between two different approaches using feature diagram. In this study, they distinguish two kinds of models: the feature models (problem space model) and software models which represents the solution-space model. The feature model is used to model the available feature and their dependencies but it does not express how a specific feature is realized. The software model specifies the techniques used for realizing the variations at the software-design level. However, we need to express a relation between model elements and features, it consists in variability modelling, a mapping between software model and features model. In [PER 08], Perrouin et al. propose to directly reference software model element from feature model to express the mapping between both spaces. The mapping between features and model elements binds the solution-space model and the model elements of each features.

3.3. Features selection

Feature model represents different alternatives of models, architecture or products. To create a specific model, architecture or product, the designer needs to select some features and use a mechanism to produce the final product. This selection of features is based on a decision model which determines which features are selected or not and can be completed by the end-user. This selection leads to the creation of the final product, step called product derivation.

3.4. Product Derivation

Heidenreich et al. [HEI 09] present different types of variability mechanisms and variability modelling approach to treat the Product Derivation (PD). Perrouin et al. [PER 08] defines the Product Derivation as “the complete process of constructing a product from Software Product Line (SPL) core assets”. According to Whitley
Variability in multi-views engineering

[WIT 96], an asset is “a description of a partial solution (such as a component or
design documents) or knowledge (such as requirement database or test procedures)”.

In a general way, we can distinguish two approaches for product derivation, a pro-
duct derivation by configuration or a product derivation by transformation:

– Product derivation by configuration This approach consists in parametrization
and/or composition of the SPL core assets. An example of product derivation is the
"staged configuration” define by Czarnecki and al. [CZA 05].

– Product derivation by transformation This approach consists in transform core
assets using MDA (Model Driven Architecture). Haugen et al. [HAU 04] present a
conceptual model for SPL engineering aligned with MDA standards.

During the derivation stage by transformation, three kinds of variability mech-
anisms can be proposed [HEI 09], positive variability which add models elements
associated with each selected features, negative variability which remove models ele-
ments that are not required according to a selection feature and modifications of Model
Elements which consists in the modification of model elements in the reference model
according to the feature selection. An example of this modification of Model elements
is component parameterization. In this case, the variability is created thanks to para-
meterized components. Then the mapping model and the selection of some features
permits the generation of values for the parameters of these components. These three
approaches need to know all the elements of the domain model used by all variants.
The difference is that positive variability supposes to create a minimal reference model
(derivation resulted model) with common or core elements contrary to negative varia-
bility where the reference model contains all models elements associated with the SPL
model. An approach to use positive variability is using it with aspect model. In this
approach a mapping model is used to indicate which aspect can be woven according
a certain selection of features. It improves modularization and facilitate maintenance
and evolution. However, if are only a few changes, it leads to a large number of small
aspect with complex dependencies among them.

In Movida project, we choose to separate feature model from models where va-
riability is applied (called base model). A mapping between feature model and base
models is added to express the model element used by each feature. The derivation
process is a model transformation that removes model elements from each view and
compose views to obtain the final product. We are working to make this tool conforms
to the future Common Variability Language (CVL) OMG Specification [OMG 09].

So, the proposed tool supports negative variability. However, positive variability
can be useful in some cases. That is why positive variability would also be added as a
future work in using Kompose to merge the different model variants.

4. Feature Diagram Tool Suite Presentation
In the context of Movida, the feature diagram tool suite is used to manage variability between a family of views. It is composed of four modules:

– Feature Diagram Editor
– Base model decorator
– Feature selection engine
– Product derivation engine

To illustrate these tools, we use an example proposed by Obeo society that models computer systems. We consider a global architecture that can be specialized by two ways, one is powered (figure 1) and another with two inputs (figure 2) for the Processing Unit element. These two ways represent variability points that we will express on our tool thanks to feature model. We choose also, as optionality the possibility of adding an additional fan.

Figure 1 – Powered Flow Example (Source MOVIDA project)

The next sections present more in details this feature diagram tool suite and the design.

4.1. Feature Diagram Editor

This section presents the choices made to develop the feature diagram editor. To start with, we present the metamodel used, then the selected graphical notation and the constraints used on the feature diagram models.
4.1.1. Feature metamodel used

On this section we will present the feature diagram metamodel used in this tool (cf 3).

*FeatureDiagram* is the root class of the metamodel. This class has an attribute *graphTypeTree* which permits to determines whether if the feature diagram is a tree feature diagram or a Directed Acyclic Graph (DAG). It also contains a list of features (class *Feature*) which are represented in the feature diagram as a node. The special
root node $r$ is identified by the reference root from \textit{FeatureDiagram} to \textit{Feature}. In the metamodel, these operators are subtypes of the class \textit{Operator}, and each feature (class \textit{Feature}) contains 0 or more operators. The class \textit{Feature} also contains a list of edges (class \textit{Edge}) allowing the construction of the set $DE$ of decomposition edges. The set $CE$ of constraint edges is represented in the metamodel by the class \textit{ConstraintEdge} and they are contained by the class \textit{FeatureDiagram}. Each \textit{ConstraintEdge} contains either a Require constraint or a \textit{Mutex} constraint. Model elements from the base model are stored directly on features with the \textit{modelElements} reference. To conclude the \textit{Attribute} metaclass defines an attribute that we can add on a feature in order to store information used to determines whether child must be selected. For example, we can add an attribute with the name of a country in a feature and choose to select one of the children feature according to this country.

Riebisch et al. \cite{RIE 02} approach limits the Feature Diagram operator to only cardinality. Even if it seems to simplify the concept of feature diagram, operands like \textit{or}, \textit{and}, \textit{xor} are commonly used and their understanding is easier. That is why we need to have operators such as \textit{or}, \textit{and}, \textit{xor}, cardinality (\textit{card}) or optionality (\textit{opt}) in our meta-model.

4.1.2. \textit{Graphical notation used on the editor}

The used graphical notation is similar to the FORM notation, except that we add an \textit{OR} operator (represented with a dark mathematical angle), a \textit{CARD} operator (represented with a white mathematical angle and its bounds) and \textit{Require} and \textit{Mutex} constraints (represented by dashed arrows, one for require and two for mutex).

4.1.3. \textit{Technologies used}

The technology used to develop this Feature Diagram Editor is Obeo Designer provided by the Obeo society which is a partner of the project. Obeo Designer \cite{OBE 10} is a commercial tool which permits to create easily Eclipse-integrated graphical editor for any DSML (GMF-like editor). This tool permits also to define different viewpoints on a given metamodel.

The figure 4 presents Obeo Designer mechanism. On the left hand-side we have the metamodel and model that permits to define the graphical application we want. These applications are presented on the right hand-side. Note that this tool permits not only define GMF-like graphical editor but also Eclipse table editor. ObeoDesigner permits to develop a DSL graphical editor on interpretation mode without generating code like in GMF (Graphical modelling Framework). However, some graphical notation cannot be obtained with simplicity. That is why some changes were made between the “conceptual” graphical notation and the real notation on the tool. So, the dark or white mathematical angles respectively for \textit{or} or \textit{xor} operator are replaced respectively by a dark or a white triangle. It is the same for the \textit{card} operator represented with a square.

The second tool used for the feature diagram editor is Praxis for expressing constraints on feature diagrams. Praxis (cf \cite{SIL 10}) is an integrated Eclipse tool developed by...
UMPC to check inconsistencies not only on a single model but also on two distinct models. It is based on Prolog. This tool permits to describe unexpected situations in rules than to a textual syntax. Indeed, it saves all actions made on the model and it can retrieve on this save any undesired pattern expressed in rules with predicates. Praxis was used to create constraints on Feature Diagram Editor.

4.1.4. Constraints used to check the feature models

Some constraints written in Praxis were defined for the feature diagram editor (such as we cannot have a mutex between a parent feature and a children for example).

4.1.5. Tool presentation

This feature diagram editor is directly integrated on Eclipse (cf figure 5). This screenshot presents the feature model used on our example. A demonstration of this tool is available on [INR 10].

This tool guarantees the validity of feature models. We can also add constraints on adding Domain Model Elements such as send a warning if a given Domain Model Element is both on a mandatory and an optional feature, or provide a way to extend our rules with rules from a given Domain Model.
4.2. **Base-Model Decorator**

The Base-Model decorator tool adds a decorator layer on a base model element graphical representation of the base-model editor when this model element is bound to an optional feature. It provides variability information directly in the base-model editor.

So, on our example on figure 6 a decorator (here an orange square) is added on the additional fan.

4.3. **Features Selection Engine**

The third tool of the tools suite is the feature selection engine. It lets the designer choose which features are required for a specific product. The first version is a textual interface implemented in Kermeta that traverses the feature model, asks to the designer the feature to select between the children of a given feature and populates a feature selection model (called resolution model).

---

1. See next section for a Kermeta presentation
To check this feature selection we will check the resolution model. This tool will permit also to select automatically require features and unselect features mutually exclusives with other features ever selected.

As a future work, we will use a form model that permits to generate automatically a customized user-interface dialogue according to the choices proposed to the designer by the feature model.

### 4.4. Product Derivation Engine

As presented before, three strategies can be used to the product derivation (e.g the selection of some features and the creation of the resulted specific viewpoint model), positive variability, negative variability, and modification of model element. We choose to support first the negative variability which consists in removing model elements referenced by unselected features. We will support also positive variability as a future work. Besides, our approach uses operational variability modelling because feature diagram editor will provide a little conditional language that selects some features according to data in the model elements. It would be a mechanism that extends...
the mechanism which could be used with the attributes. This mechanism permits to select features or not according to the value of this feature. With a given feature diagram and a resolution model completed indirectly by the end-user we can create a new viewpoint model as a resolve model. Constraints will be checked on variability model and resolution model. As a future work the tool may load and check constraints on base model. These constraints would be expressed outside the tool.

On our derivation approach we deal with four models: base model (model that represents the whole product line), variability model (in our case feature model: represent available choices and their relation with the base model), resolution model that stores user features choices and resolved model that correspond to a specific product of the product line represented by the base model. Base model and Resolved model are conform to the same metamodel. This is the future OMG standard CVL (Common Variability Language) approach (presented on figure 7).

The derivation engine is currently implemented in Kermeta. Kermeta is a MDE platform designed to specify constraints and operational semantics of metamodels [MUL 05]. The MOF [OMG] supports the definition of metamodels in terms of packages, classes, properties and operations but it does not include concepts for the definition of constraints or operational semantics. Kermeta extends MOF with an imperative action language for specifying constraints and operation bodies at the metamodel level. A complete description of the way Kermeta is designed can be found in [MUL 05]. One of the key features of Kermeta is the static composition operator, which allows extending an existing metamodel with new elements such as properties, operations, constraints or classes. This operator allows defining these various aspects in separate units and integrating them automatically to the metamodel. The compo-
sition is done statically and the composed model is typed-checked to ensure the safe integration of all units. This mechanism makes it easy to reuse existing metamodels or to split metamodels in reusable pieces. It also provides flexibility. For example, several operational semantics can be defined in separate units for a single metamodel and then alternatively composed depending on a particular need. This is the case for instance in the UML metamodel when several semantics variation points are defined. This composition feature is used to specialized the product derivation engine for each DSML.

On this example, we can launch the Product Derivation directly on the feature model editor and select features through the Eclipse console like in the figure 8.

Figure 8 – Product Derivation Engine

5. Conclusion

This paper details the features and the architecture of a tool-suite to model variability into model-driven viewpoint engineering. This tool-suite proposes four modules to model variability of views using a feature diagram, to select features required for a specific product, to derive a product and to visualize variability information directly inside the base-model editor. Feature diagram editor is itself designed as a viewpoint. It has its own meta-model, its own graphical representation and share constraints with other viewpoints. This editor allows to specify the variability separately from the base-model. It has also a plug-in that adds constraints to guarantee the correctness of feature models. The product derivation engine currently supports the negative variability. Experiments are currently made by Thales on the feature diagram editor. This tool suite is also evaluated by several academics. As a future work, in the context of the Movida project, product derivation will be completed to be compatible with the new Common Variability Language (CVL) OMG’s standard. We will also working on extensible derivation engine to support positive variability and modification of model elements as defined in this standard.
6. Bibliographie


