Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 152 (2006) 161–173

www.elsevier.com/locate/entcs

Requirements Variability Support Through MDA™ and Graph Transformation

Javier Pérez [1], Miguel A. Laguna[1], Yania Crespo González-Carvajal[1] and Bruno González-Baixauli[1]

Departamento de Informática
Universidad de Valladolid
Valladolid, Spain

Abstract

One of the most important factors of success in the development of a software product line is the elicitation, management, and representation of variability. Feature models, are used as a key artifact to express requirements variability and are the basis for the domain architecture design. In this context, this article explores the possible advantages of Model Driven Engineering (MDE) and shows an automated transformation from the feature model to the architecture model. This transformation is understood as a graph transformation process because it offers a natural way to represent model transformations. The transformation is applied by the definition of a simple context-sensitive graph grammar where production rules are obtained from metamodels of both feature and architecture models.

Keywords: MDE, Requirements variability, Feature Model, MDA™, Graph Transformation, Layered Graph Grammars

1 Introduction

Product lines (PL) have become the most successful approach in the reuse field, due to the combination of coarse-grained components, i.e. software architectures and software components, with a top-down systematic approach, where the software components are integrated in a high-level structure. However, product lines is a very complex concept that requires a great effort in

[1] Email: {jperez,mlaguna,yania,bbaixauli}@infor.uva.es

doi:10.1016/j.entcs.2005.10.023

both technical architecture definition, development, usage and instantiation [2,4] and organizational business view [1] dimensions. In addition, the standard proposals of the software development process traditionally ignore reuse issues, in spite of their recognized advantages [12]. Our proposal is to introduce a reuse approach based on product lines that requires less investment and presents results earlier than more traditional product line methods [14]. This proposal incorporates the best practices in reuse approaches, mainly from the domain engineering process, into conventional disciplines of the application engineering process. The first point we focus on, as it is one of the most critical, is the elicitation and analysis of variability in the product line requirements. We have explored two techniques: Goal Oriented Requirements Engineering and Model Driven Architecture (MDA™). The goal approach to variable requirements elicitation has been treated in detail elsewhere [8,9]. In this paper we focus on Model Driven Architecture.

Model Driven Architecture (MDA™) was introduced by the Object Management Group (OMG) and is based on the Platform Independent Model (PIM) concept. The PIM is a specification of a system in terms of domain concepts and with independence of platforms (e.g. CORBA, .NET, or J2EE) [10]. The system can then transform the PIM into a Platform Specific Model (PSM) [10]. As the main strength of MDA™ is the manipulation and transformation of different models and feature and goal models are introduced in our process, it is worth exploring the relations of these models with UML conventional models in the MDA™ context. Transformations from feature and goal models to UML class diagram are given. The transformation rules are explained and later modeled as graph transformations. Graph rewriting rules are specified on metamodels. The starting graph is an instance of a feature metamodel and the final graph is an instance of the UML metamodel.

The rest of the paper is as follows: The next section (Section 2) discusses the benefits that MDA™ can bring to the product line approach and address the transformation from feature and goal models to an UML class diagram representing part of a simple framework design. Later, Section 3 shows how to implement the transformation with graph and graph rewriting formalism and tools. Section 4 concludes the paper and proposes additional work.

2 MDA and Product Line Requirements Engineering

The OMG site refers to some successful experiences with MDA™. Yet, with respect to the application of MDA™ to product line development, the pertinent question is: What degree of real freedom exists at the time of creating a PIM? As a typical example, in the book by Kleppe et al. [13] a translation of

a PIM is a set of three PSM which are predefined concrete solutions: a PSM based on Web technology, another supported by java beans technology and the third based on relational databases. Another approach, the executable UML paradigm [15] is specific to a certain kind of system and requires a precise definition of the classes and operations (using an action semantic language very close to conventional code). Nevertheless, when a product line requirements model is specified, its creation is accompanied by other requirements of quality, security, etc. These non-functional requirements influence the type of architectural solution and technologies that must be applied. The assumption is that, in the previous examples, there is a hidden set of NFR that are not specified in the PIM. In spite of these inconveniences, it is worth analyzing the possibilities that the MDA™ ideas can bring to this field. Essentially we are searching for an (ideally automated) derivation of an optimal specific product in a product line, while taking into consideration functional and non-functional requirements and using the goals/soft-goals and feature models and their correlations as the starting point. A set of transformations between these models can actually be carried out.

The Product Line Requirements Engineering discipline includes several activities. The main activity involves the specification of the domain model, which consists of the domain features. The design of a solution for these requirements constitutes the architectural asset base of the product line (typically implemented as an OO Framework).

Later, in the application engineering process, an application model must be derived from the domain model. In this process, alternative concepts are selected based on customer functional and non-functional requirements. This activity is essentially a transformation process where a set of decisions taken by the application engineer generates the initial feature product model and, consequently, via traceability links, the initial architecture of the product. The variation points are selected on the conceptual level on the basis of a rationale provided by functional and non-functional requirements.

The novelty with MDE (particularly MDA™) is the possibility of the automation of the transformations that specify how instances of the domain feature model are converted into a working application. A pre-requisite of the applicability of MDA™ is to have a metamodel of each technique. In Figure 3, a feature metamodel is presented. The transformation definition can be seen in its more mature state as a compiler for a domain specific language. The feature/goal models combination would be compiled into a working application using the transformation definition, the asset base and the customer requirements.

In [4] MDA™ is presented as an approach to derive products in a specific

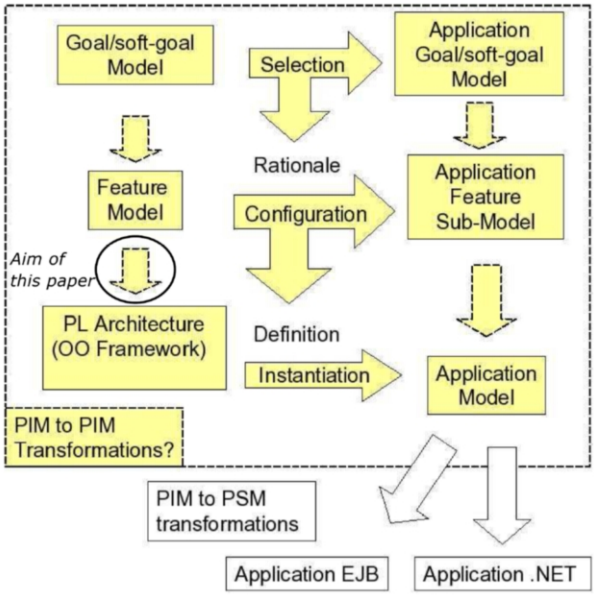

Fig. 1. Product line engineering and MDA™ and the scope of this study. Left and right parts of the figure refer to Product Line and Application processes respectively

type of product lines, configurable families. The authors main idea is that a software system that is specified according to the MDA™ approach is a particular case of product line where the most characteristic variation point consists of products that implement the same functionality on different platforms. The choice for the alternative platforms is a variation point in such a product line. This variation point can be separated from the specification models and managed in the transformation definition itself. The main benefit of MDA™ compared to traditional development, is that the management of the platform variation point is handled automatically by the transformation step and is not a concern for the product engineer.

However the final platform for a product is not the only variation point that needs to be managed in a product line. The various product line members differ in both their functional and non-functional requirements.

The central question is if MDA™ can easily accommodate these variable requirements by adding the information that specifies places where alternative concepts can be selected. Selection of different concepts from this domain model then results in different PIMs, specific to an application, which, provided that the adequate transformation definitions are implemented, can be automatically transformed into PSMs using the MDA™ approach. The general schema is presented in Figure 1.

From an application model to specific platforms (.NET or EJB), a conventional PIM to PSM transformation can be defined and this is the only typical

MDA™ transformation, where the initial application model obtained from the feature model and the asset base (and manually completed) is transformed to a specific PSM or set of PSMs. The rest of the models in Figure 1 can be considered as PIMs.

These PIMs can be related via automated or non-automated transformations. The possibilities MDA™ offers must be examined in detail. There are basically two kinds of transformation:

- **Horizontal:** selection of goals/soft-goals combination, feature model configuration, and framework instantiation
- **Vertical:** PL Goal model to PL Feature model transformation, PL Feature model to PL Architecture (Domain framework) transformation, and the parallel Application equivalents.

Some feature models can incorporate platform or context information (such as variability points) but as we use the soft-goal model to express non-functional variable requirements, the feature model is basically functionally oriented. Inspecting the Figure 1 and from the horizontal point of view, we can extract some conclusions, independently of the automation degree of the transformations.

The conventional configuration step of a feature model consists of imposing a set of constraints which originates the selection of a sub-graph of features, possibly with some alternative variants deferred to execution time (Czarnecki differentiates configuration from specialization mechanisms of derivation of feature sub models [7] but we only contemplate configuration as an horizontal transformation for the sake of simplicity). There are several kinds of tools to select the variants, such as wizards or graph-like languages and their use guides the instantiation of the particular application model. The difficulty is that the combination of features must be decided by a domain expert based in his experience and not in objective data.

Using our complementary goal/soft-goal model fulfills a twofold purpose, it allows the application engineer to deduce (if the traceability links are carefully established):

(i) **what features are needed** to reach the selected goals (or functional requirements)

(ii) **which is the optimal set of goals/features** in the context of a set of soft-goals (or NFR) of a determined priority that provides the rationale of the selection

In practice, this supposes a rise in the abstraction level of the variants selection process, making the selection in the requirements level instead of in

the feature level. As a conclusion, these horizontal transformations can be automated, but not in the MDA™ sense. This line of work can be supported by the tool described in [8] and, despite its scalability problem, the obtained results are promising. In the rest of this section we will focus on the vertical possibilities of Figure 1, basically the steps from the PL Goal model to the PL architecture. The Application Goal, Feature and Architecture models can be better derived as instances of their PL corresponding models (the horizontal transformation), instead of using a vertical approach.

The relation between PL Goal and Feature models cannot easily be considered as an MDA™-like transformation because of the different objectives and building methods. Goal models determine the variability of the different ways to achieve these goals (expressed as a tree of sub-goals and tasks that operationalize these goals), while feature models separate the common from the variable part of the systems. This characteristic implies that, until this moment, the two models (three, if we consider the soft-goal model) must be built manually, but not independently, by the domain engineer. As a working hypothesis, a constraint imposing that a Task must be implemented only by one Feature will facilitate the traceability and the selection of components from a goal configuration and also a possible derivation of an initial PL Feature model from the PL Goal model.

From the point of view of the PL Feature model to PL Architecture transformation, the method we have chosen is based on the metamodel mapping approach [6]. The work consists of defining a set of transformations between the elements of variability in the feature models and the architectural solutions (really each kind of variability in the feature model can be implemented by more than one technique [5]). An example can be seen in Figure 2: a feature model is transformed into a model that represents part of a simple framework design.

The way to define a transformation is to select an element of the feature metamodel and give one or several equivalences in the UML metamodel. This implies that an annotation is needed in the feature model to select one of the possible design mechanisms. As we need a precise definition of the metamodels, the first consideration is to answer the question about the metamodel compatibility of these different models. It is clear that the Application metamodel is the UML metamodel. In the case of the PL Architecture, the UML metamodel is also used. (Some authors propose extensions or profiles to complete the information about the variability that can be used to support traceability.) The Feature models (and sub-models) are built using other concepts but several studies have specified different metamodels using MOF. We have explored these metamodels and conclude that the election influences greatly

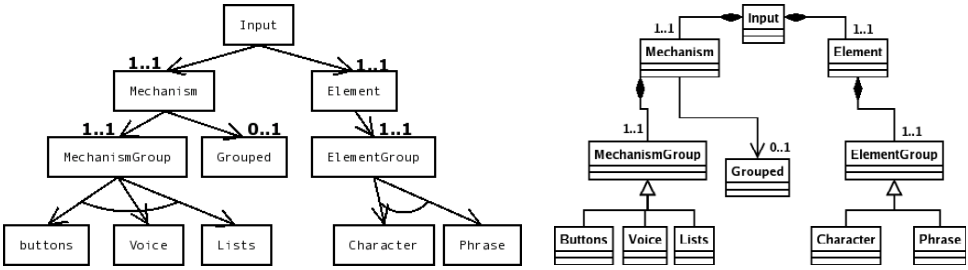

Fig. 2. Feature model in the domain of communication for handicapped people and a simple solution using composition, association, and specialization

the transformation process. The Massen proposal [17] was used initially but finally the recently proposed by Czarnecki et al. [7], has been selected because the simplicity of the related transformation. In this approach, the distinctive property of the relationships is the cardinality. In the metamodel of Figure 3, the relationships are implicit and the source of the transformation must be the cardinality attribute of the features and group of features.

3 An example transformation: PL Feature model to PL Architecture model

The models involved in the transformation to be explored, the Feature metamodel and the UML partial metamodel, are shown in Figures 3 and 4 respectively. Transformation rules are graphically expressed using the latest QVT submission syntax [11] in Figure 5. The main interest of this transformation is that once it's defined, the generated framework can be used to automatically derive the application model by selecting the desired features, as mentioned above.

The strategy is to order transformations, from root to leaf nodes of the feature tree diagram, considering each feature subtype involved. First, Feature Model is translated to a Namespace. Afterwards Root Feature nodes are converted, and cyclic transformations of Solitary Features and Feature Groups are finally carried out.

Once the metamodels and transformation rules are selected, a formal foundation is needed to support their implementation in CASE/MDE enabled tools. Since the models involved are represented with graphs, graph transformation formalisms support these transformations straightforward. Contextual Layered Graph Grammars (CLGGs [3]) are chosen because they allow to specify a rule ordering. In CLGGs, the set of available rules are divided in different subsets classified in ordered layers. To transform a graph, rules must be executed layer by layer. Before a rule belonging to an upper layer can be applied,

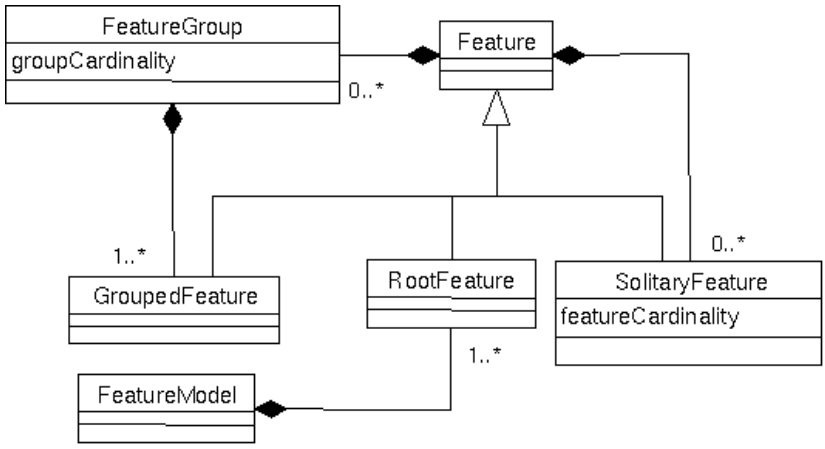

Fig. 3. Feature simplified metamodel, adapted from Czarnecki et al.

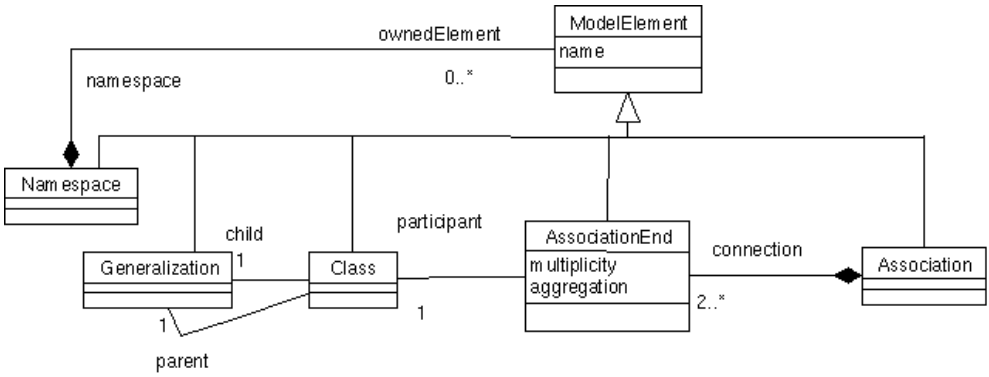

Fig. 4. partial UML metamodel

there can not be an applicable rule available on the lower layer. We use a tool called AGG [16] that supports CLGGs and Attributed Graph Grammars (grammars that allow the use of attributes, cardinalities, for example). AGG can be used without its GUI, as a separate graph transformation engine, that can be integrated into dedicated tool. This makes AGG the most adequate tool to implement our MDE transformations.

QVT rules have been implemented in AGG. Each transformation rule definition is translated to a set of rules that implement it. Different layers are assigned to each set of rules related to each QVT definition. This guarantees that rules are applied in the same order established by the metamodel mapping definition.

Graph transformation rules consist of three parts: a left side, or positive application condition, a right side, and a set of negative application conditions. The left side of the rule states a morphism that must be found in the graph

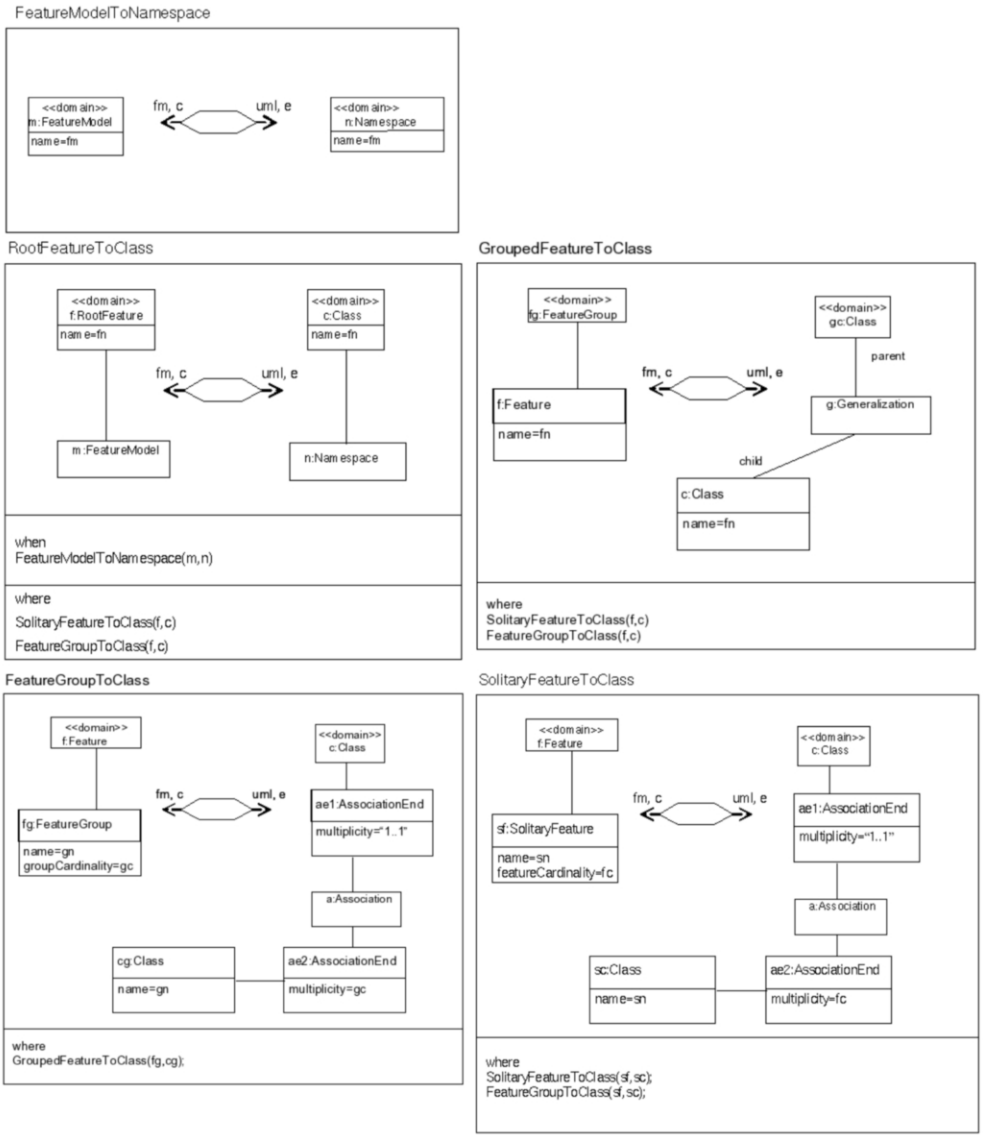

Fig. 5. Transformation definition of a Feature model into a UML metamodel

to apply that rule. The right side of the rule defines which elements of the left side are deleted, which ones are preserved and what new elements must be created in the resulting graph. The negative application conditions define subgraphs that must not exist in the graph to be able to apply the rule. Some additional restrictions must be considered in the graph transformation rules. For example, deletion of nodes are not allowed to leave dangling edges in the resulting graph.

Because of these restrictions, each single QVT definition must be translated

to a set of graph transformation rules. This set can be seen as a three-stage transformation process.

(i) **Create classes:** One rule is needed to create the PL architecture classes from the corresponding PL Feature node. Negative application conditions must be declared to assure the execution of exactly one transformation per feature.

(ii) **Move children:** One or more rules are needed to move all links between the feature node and its children nodes. Relationships are moved from the feature node to the class node. One rule per children node type is defined.

(iii) **Deletion of feature node:** When no more "move children" rules can be applied, the feature node is deleted from the graph. No node with children links can be deleted because of the dangling edge restriction.

Layering rules let us to define the same execution order defined in the QVT transformations. So *FeatureModelToNamespace* rules are assigned layer 0, *RootFeatureToClass* rules correspond to layer 1, *SolitaryFeatureToClass* to layer 2, *FeatureGroupToClass* to layer 3, and *GroupedFeatureToClass* to layer 4. This transformation process is stated to cycle through the last three rules. This is also managed by AGG, that allows to "loop over layers".

As an example, we show in Figure 6 the set of rules that implement the *RootFeatureToClass* transformation definition. To simplify the example, a single link type is used for each model, and cardinalities are not used. Both can be easily added as node/edge attributes. In **rule 1**, for each *RootFeature* node the corresponding class is created. Notice that *Namespace* has already been created during application of layer 0 rules. Relationships between *RootFeature* class and *Namespace* are also created. A negative application condition (left most part of the figure), is defined to avoid creating more than one class for each *RootFeature* node. In **rules 1 and 2**, links from children nodes are moved to the translated *RootFeature* class node. One rule per children node type is needed. In **rule 3**, a single *RootFeature* node appears on the left side, and the right side is empty. The application of this rule leads to deletion of the *RootFeature* node.

The example previously introduced in Figure 2 is used in Figure 7 to show how the feature model is represented as a graph (the starting graph). Once the defined rewriting rules are applied, we obtain the resulting PL Architecture graph in Figure 8 as an instance of the partial UML metamodel.

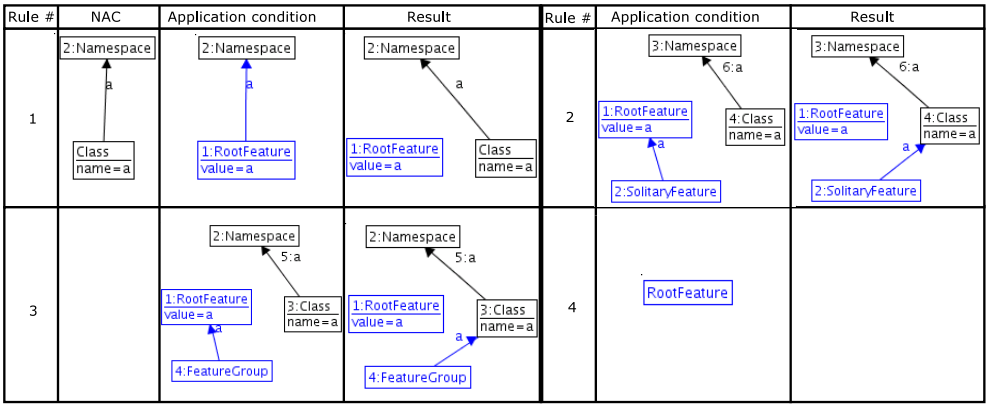

Fig. 6. Graph transformation rules for *RootFeature* QVT

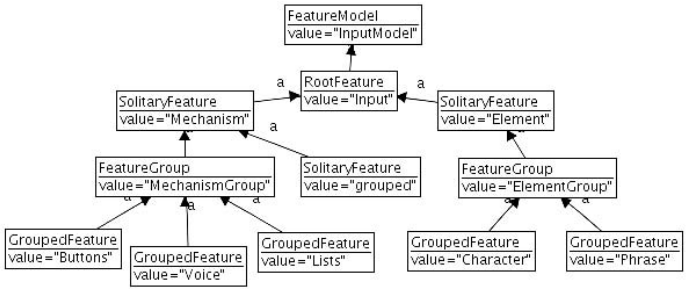

Fig. 7. Initial graph: a simple feature metamodel instance

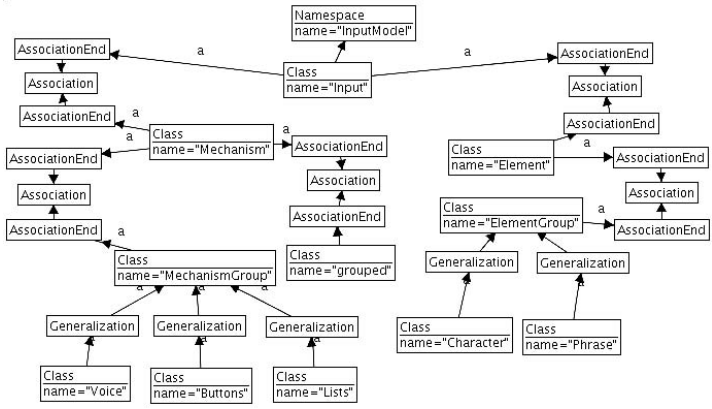

Fig. 8. Resulting graph: the corresponding framework architecture

4 Conclusions and Future Work

This paper presents an approach on the use of MDE to support requirements variability. More precisely, it explores the transformation of feature models into architecture models. Transformation are defined in terms of QVT map-

ping rules that link feature and UML metamodels elements respectively. This transformation has been formalized and implemented through graph transformations. Representation of models to be transformed, have been defined as their metamodels instances with directed typed graphs. Transformation rules have been implemented through Contextual Layered Graph Grammars [3]. The process has been supported by the graph rewriting tool AGG which can be used without its GUI and can be integrated as a graph transformation engine in the future in our own MDE tools.

In this article, the possibilities provided by new technologies such as MDA™ in the process of requirements elicitation and analysis are discussed in the context of product line development. The solution that we envision would happen in several steps:

(i) To separate different aspects of the requirements in an explicit form, using goals and soft-goals models (as PIMs) to finally build the product line feature model.

(ii) To transform this set of PIMs into a new PIM that represents the initial PL Architecture (in the form of an object-oriented framework) and complete it manually with design details (but saving the traceability links with the goal/feature variation points).

(iii) To derive an optimal sub-graph of features to solve a concrete problem in the product line, using the goal/soft-goal model as a reference (and with a tool like that described in [8]).

(iv) To build the architectural PIM for the new application from the product line architecture as a framework instantiation, using the goal/feature submodel as a guide.

The most immediate pending work comprises the inclusion of explicit traceability in the transformation specification and implementation. This approach implies in consequence the enhancement of the supporting meta-models.

Acknowledgement

This work has been supported by the Spanish MEC/FEDER (TIN2004-03145).

References

[1] L. Bass, P. Clements, P. Donohoe, J. McGregor, and L. Northrop. Fourth product line practice workshop report. Technical Report CMU/SEI-2000-TR-002 (ESC-TR-2000-002), Software Engineering Institute. Carnegie Mellon University, Pittsburgh, Pennsylvania 15213 (USA), 2000.

[2] J. Bosch. *Design & Use of Software Architecture. Adopting and Evolving a Product-Line Approach.* Addison-Wesley, 2000.

[3] Paolo Bottoni, Gabriele Taentzer, and Andy Schrr. Efficient parsing of visual languages based on critical pair analysis and contextual layered graph transformation. In *VL '00: Proceedings of the 2000 IEEE International Symposium on Visual Languages (VL'00)*, page 59, Washington, DC, USA, 2000. IEEE Computer Society.

[4] P.C. Clements and L. Northrop. *Software Product Lines: Practices and Patterns.* SEI Series in Software Engineering. Addison-Wesley, 2001.

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

[6] K. Czarnecki and S. Helsen. Classification of model transformation approaches. In *OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[7] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration through specialization and multi-level configuration of feature models. *Software Process Improvement and Practice*, 10(2), 2005.

[8] B. González-Baixauli, Leite J.C.S.P., and J. Mylopoulos. Visual variability analysis with goal models. In *Proc. of the RE 2004*, pages 198–207. Kyoto, Japan, IEEE Computer Society, Sept. 2004.

[9] B. González-Baixauli, M.A. Laguna, and J.C.S.P Leite. Análisis de variabilidad con modelos de objetivos. In *VII Workshop on Requirements Engineering (WER-2004). Anais do WER04*, pages 77–87, 2004.

[10] Object Management Group. *MDA Guide Version 1.0*, 2003.

[11] Object Management Group and QVT-Merge Group. *Revised submission for MOF 2.0 Query/View/Transformation version 2.0.* Object Management Group doc. ad/2005-03-02, 2005.

[12] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process.* Object Technology series. Addison-Wesley, 1999.

[13] A. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise.* Addison Wesley, 2003.

[14] MA. Laguna, B. González-Baixauli, O. López, and F.J. García. Introducing systematic reuse in mainstream software. In *IEEE Proocedings of EUROMICRO'2003, Antalya, Turkey*, 2003.

[15] S.J. Mellor and M. J. Balcer. *Executable UML A foundation for the Model-Driven Architecture.* Addison Wesley Professional, 2002.

[16] Gabriele Taentzer. Agg: A graph transformation environment for modeling and validation of software. In *AGTIVE 2003*, pages 446–453, 2003.

[17] T. von der Massen and H. Lichter. Requiline: A requirements engineering tool for software product lines. In *Software Product-Family Engineering, PFE 2003*, pages 168–180. Siena, Italy, LNCS 3014, Springer-Verlag, 2003.