

Feature Patterns and Product Line Model Transformations

Miguel A. Laguna, Bruno González-Baixauli

Department of Computer Science, University of Valladolid,
Campus M. Delibes, 47011 Valladolid, Spain
{mlaguna, bbaixauli}@infor.uva.es

Abstract: Feature models are the basic instrument to analyze and configure the variability and commonality of a software product line. But feature models embody various different variability facets (structural, behavioral, non-functional, etc.). Features, used as core model, must be completed with other techniques (i.e. goals or UML models) to fulfill these variability aspects. This approach allows us to proceed in several steps, using the appropriate paradigms in each phase. The global picture is a sequence of model transformations from goal/requirements to features and from both to the architecture. In this context, this article aims to identify patterns in the feature models and their relation with the corresponding architectural counterparts (class and use case diagrams). The work is completed with the definition and implementation of meta-model based transformations between these models. The existence of a feature pattern catalog and the associated transformations make the automated creation of models and traceability links possible, enhancing the productivity of the development process of product lines.

Keywords: feature models, software product line, model transformation

1 Introduction

The development of software product lines (PL) faces many technical and organizational trends, in spite of its success in the reuse field [2, 5]. On the requirement level, one of the key activities is the specification of the variability and commonality of the PL. The design of a solution for these requirements constitutes the PL domain architecture. Later, the concrete product architecture must be derived for each variant in the family. In this process the functional and non-functional customer requirements are used for choosing among alternative features and, indirectly, via traceability paths, the architecture of the product [2].

Feature Oriented Domain Analysis (FODA) [18] proposed features as the basis for analyzing and representing commonality and variability of applications in a domain. A feature represents a system characteristic realized by a software component. There are four types of features in feature modeling: Mandatory, Optional, Alternative, and Or groups of features. However, although its effectiveness has been proven in many projects, these models are more oriented to the solution than to requirements. On the other hand, non-functional requirements (NFR) and platform specific aspects must be taken into account and other models can express these aspects better (we use UML models and goal oriented techniques). Consequently, the use of the feature diagrams as a monolithic tool over-simplifies and limits the potential of the technique.

The origin of the problem is that FODA feature models try to cover different variability facets simultaneously: they represent structural, behavioral, non-functional, and platform variability. We consider that other well known techniques, different from feature diagrams (such as goal and UML models) are better at expressing different aspects of variability, while the feature model can be the central piece of the puzzle that connects the rest of the models. Taking into account the different categories of features, we have proposed to analyze them separately [20]:

- Goal models (essentially hard goal) represent the intention of the PL, i.e. the high level objectives the family of applications must solve. The *soft-goals* represent the non-

functional characteristics. These can be used with the hard-goals contribution information to configure the optimal solution for a concrete application in the PL [11].

- Feature models represent the functional variability. They connect the goals with the UML models. Two main categories of features are included: structural and behavioral (services and operations) features.
- UML models organize the architecture specification, including structural and behavioral system requirements of the PLs, connected with the feature model.

The aim was to analyze the product line requirements using the appropriate model for each different variability aspect. The development of a PL can be seen as a sequence of model transformations from goals/*soft-goals* to features and from both to the initial architecture (a set of UML models), tracing the relationships between these models.

We have manually developed some case studies of PLs using this approach¹. However, the productivity (and the complexity in non-trivial product lines) requires the construction and derivation of the different models to be automated. As the main strength of the Model Driven Engineering (MDE) paradigm is the manipulation of models, we propose to use this approach to define these transformations. This approach requires the establishment of precise rules, using meta-modeling and transformation tools. To establish these rules, the recurring patterns in the feature models must be discovered and correlated to UML structures. In this context, the next Section identifies the basic patterns that can be found in feature models. The result is a catalog of feature patterns and their corresponding structural and behavioral UML models. Section 3 shows the application of the MDE approach automating these transformations, using meta-modeling and QVT [26]. Section 4 presents related work and Section 5 concludes the paper and proposes additional work.

2 Feature Patterns

In a product line development process, after the goals and feature models are established, several structural and behavioral UML models must be built (hopefully using a catalog of commonly used derivations from feature to UML models). A revision of the literature has shown it to be naive to try to make a simple and univocal transformation from feature models to UML diagrams. Therefore, we have adopted a pragmatic and multi-view approach as stated in the introduction: separating the different categories of features and analyzing them with different techniques. In this section, we use a selection of examples extracted from a large case study described in [23], using feature models, class models and activity diagrams. It is an electronic commerce product line (manually developed) and includes a great number of optional features and, as a consequence, a large number of potentially derived products. The feature tool used is the *fmp* eclipse plug-in developed in Waterloo University [1] and an adapted version of a conventional CASE tool.

We differentiate two kinds of transformations: the structural information of the feature models is mapped to class diagrams and the behavioral features are connected with use case diagrams.

Consider the simplest situation. An AND feature construction that represents structural information can be directly transformed into classes and attributes (Figure 1). In our approach, based on [9], the features are typed. The type can be a simple type or the default FEATURE type itself. The key aspect is that the leaves of the tree can be features with information about simple types and in these cases a simple typed feature is mapped into an attribute of a parent class.

If the leaf is not an atomic feature (or is simply an intermediate node in the tree) a little more complicated solution is required. The mapping consists of creating a class that represents the feature (more details will be added during the later refinement of the structural model). The mandatory and optional features imply a 1..1 or 0..1 composition relationship respectively. Figure

¹ available at <http://www.giro.infor.uva.es> as student term projects

1 includes some examples of both circumstances: *CreditCardInfo* is a class that represents optional information in the class model; *LoginCredentials* represents compulsory information.

This enables the identification of several simple patterns in feature models:

- The presence of a mandatory feature of FEATURE type originates a class that is associated (with a 1..1 multiplicity) with another class that represents the parent feature.
- The presence of an optional feature of FEATURE type originates a class that is associated (with a 0..1 multiplicity) with another class that represents the parent feature.
- The presence of a mandatory feature of a simple type (INTEGER, STRING, DATE..., i.e. any type different from default FEATURE type) originates an attribute in a class that represents the parent feature. The presence of an optional feature defined by a simple type originates an optional attribute (represented by the UML attribute multiplicity information) in the class that represents the parent feature.

Summing up, the structure of classes/attributes connected by a 1..1 or 0..1 multiplicity association or composition relationship reflects the original structure of the features.

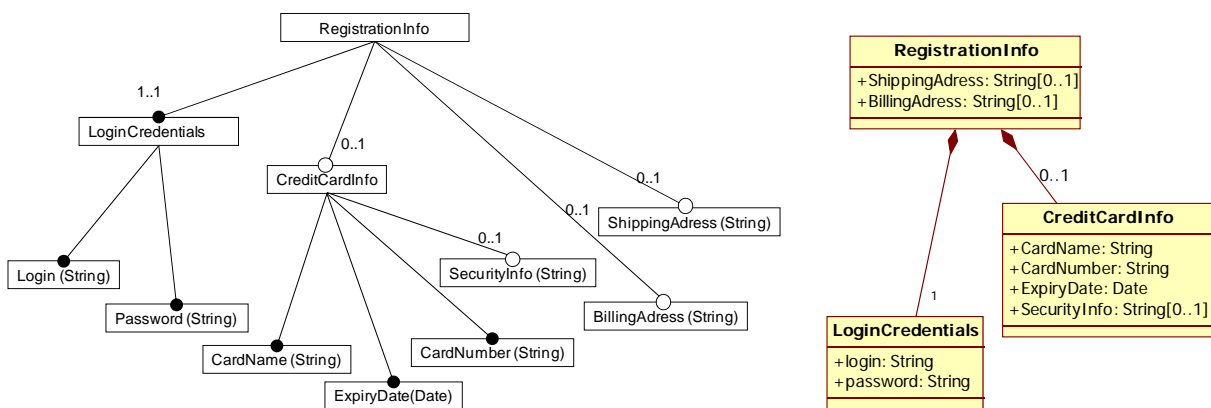


Fig. 1 Structural feature model fragment and the corresponding UML model equivalent

The common architectural equivalence of a set of grouped features (alternative and OR groups) is based on generalization. In fact, a combination of composition and generalization relationships is needed to differentiate alternative from OR patterns. Both variants have a common part in the corresponding class diagram structure: a composition relationship indicates whether the selection of one or several sub-features is compulsory (minimum multiplicity equal to or greater than one) or optional (minimum multiplicity equal to zero). The composition maximum multiplicity differentiates the OR relation (maximum multiplicity greater than one) from the pure alternative relation (maximum multiplicity exactly equal to one).

Figure 2 shows examples of both variants. The *CheckOut* feature always has a *CheckOutType*, but this must be (in a concrete application) a *Registered CheckOut* or, alternatively, a *Guest CheckOut*. The two variants cannot be implemented simultaneously in a concrete application. This is reflected in the composition relationship shown in the same Figure: only one instance of type *CheckOutType* can be connected to a *CheckOut* instance. However, the variant of the *PaymentTypes* feature implies that more than one feature can be selected (but one of these must always be the *CreditCard* payment type). The multiplicity 1..3 of the composition relationship states this possibility. Finally, the *PaymentGateways* group of features are optional and compatible (the multiplicity 0..2 clarifies the possibilities).

These patterns conform to the literature. However, we can appreciate that some basic information is lost. For example, it is necessary to show that the payment type *DebitCard* or *ElectronicCheque* can be used in some applications or not, but each application in the PL must have the *CreditCard PaymentType*. This is not shown in Figure 2, and this representation must be modified to clarify these differences.

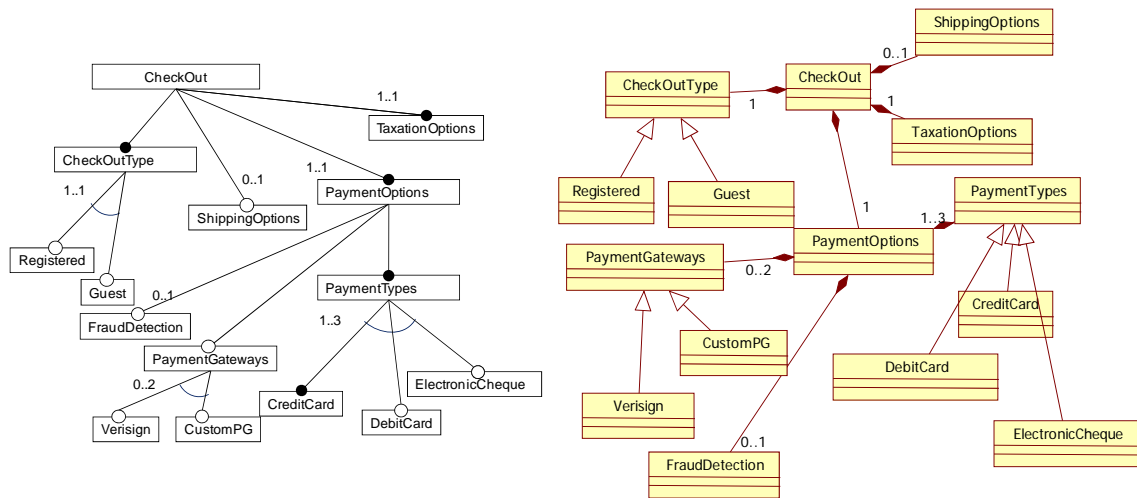


Fig. 2 Structural feature model fragment with examples of alternative and OR feature constructions and the corresponding literature based architectural solution

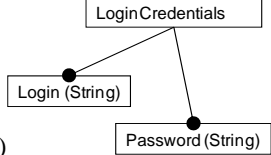
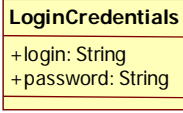
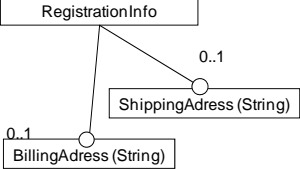
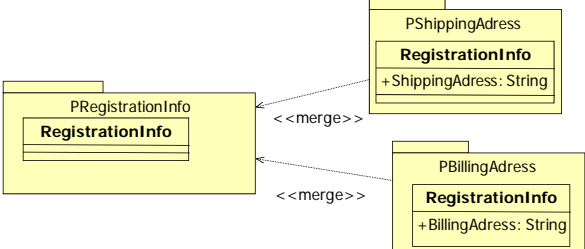
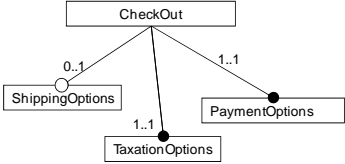
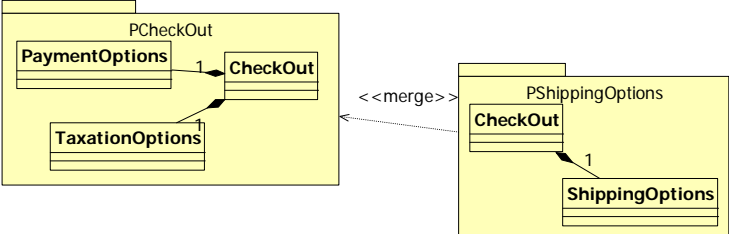
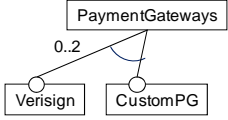
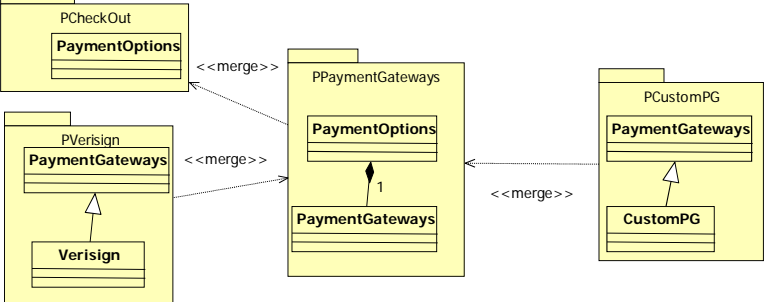
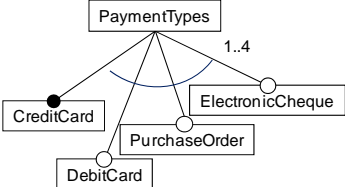
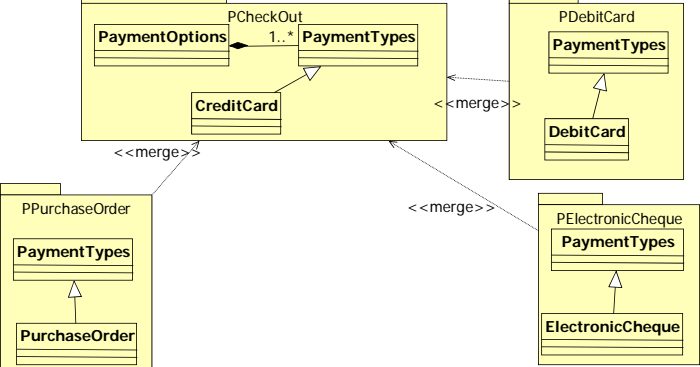
In a previous work we proposed to separate the representation of variability of the PL completely from the variability of the concrete applications [21], using the UML package merge mechanism to add details to the models in an incremental way. The `<<merge>>` relationship indicates that the contents of two packages are combined. It is used when elements in different packages have the same name and represent the same concept, which is extended incrementally in each separate package. Selecting the desired packages, it is possible to obtain a tailored definition from all the possible ones. Although the examples in UML focus on class diagrams, the mechanism can be extended to any UML model, in particular use cases and sequence diagrams.

This mechanism provides a clear traceability between feature and UML models. Each optional feature is described by an optional package of the PL that will be selected or not, in function of the concrete configuration of features. The process consists of establishing, for each UML model, a base package that embodies the common part of the PL. Then, associated to each optional feature, a new package is built, so that all the necessary changes in the model remain located. This package is connected through the `<<merge>>` relationship with its base package in the exact point of the package hierarchy. The direction of the relationship expresses the dependence between packages: the base or merged package can always be included in a specific product, the receiving package is an extension of the base package and can only be included if the base package is also selected. This is exactly the way the expert decides which features are included or not during the configuration process, and must be directly reflected in the configuration of packages. A modified version of the feature patterns is imaginable, combining the literature based structures with the package merge mechanism.

The representation of the PL structural model we use is shown in Table 1, which presents some examples of the package based version of the feature patterns equivalences with structural models. This second version is preferable, as it removes any ambiguity about the multiplicity semantics and is directly translatable to code.

The example 5 is clarifying: All the applications in the PL have the possibility of Credit Card payment but some of them have two or three additional possibilities of payment. This doesn't presume which type of payment (or combination) will be selected by the final user in each system execution. The semantics of cardinality in feature models is different from the multiplicity semantics in class diagrams and is actually related to the package organization of the PL architecture. A collateral benefit is that the traceability between features and packages is registered automatically.

Table 1 Examples of basic structural features and their transformation to package combination of class diagrams

Feature Pattern	Class Structure (package merge variant)
<p>(1)</p> 	
<p>(2)</p> 	
<p>(3)</p> 	
<p>(4)</p> 	
<p>(5)</p> 	

A similar argumentation can be used if we want to find the behavioral equivalences of feature patterns. We base the UML behavioral model on use case diagrams (obviously combined with the package merge mechanism) with intensive use of include and extend relationships. Mandatory behavior is modeled with included use cases and optional behavior with extension use cases. But

optional behavior has, as in structural models, two meanings. Using the same example, the details of payment steps can be different in an implemented application in the PL because it is possible to pay by cheque or credit card (and then we use the extend relationship). However, any application always requires the credit card payments but not the cheque possibility (this second behavior is in an extension use case inside an optional package). This approach leads us to Table 2, basic examples of patterns (optional/mandatory and include/extend combinations). The name of use cases can be modified as a part of the transformation (using a convention as “manage X”) but this is not relevant as the internal structure of each package is only a first cut that must be manually refined (in contrast, the package structure must be preserved).

Table 2 Basic behavioral features and their transformation to package combination of UML use case diagrams

Feature Pattern	Use Case diagrams (package merge variant)
<p>(1)</p>	
<p>(2)</p>	

3 Transformations of feature patterns

The complexity of the models forces us to automate the transformations in order to successfully adopt the product line paradigm. As the method we have chosen is based on the meta-model mapping approach [8], our work consists of defining a set of transformations between the feature patterns and the architectural UML constructions (both defined in terms of the corresponding meta-model), based on the informal finding of the previous section.

The way to define a transformation is to identify an element (or more exactly a combination of connected elements that satisfy certain constraints) of the feature model and give the two equivalences in terms of the UML meta-model. As the election of the meta-model greatly influences the transformation process, we evaluated several possibilities specified in the literature. The meta-model proposed by Czarnecki *et al.* in [9] has finally been selected because of the simplicity of the transformations. In this approach, the distinctive property of the relationships is the cardinality. In a previous work [21], we defined and implemented the structure of packages that represent the architectural vision of the PL, facilitating the traceability of the models. The transformation is now completed, filling the packages with classes, attributes, and relationships (and in parallel, use cases and extend/include relationships).

The transformation from feature into structural models (Table 1) implies:

- a) Transform the Feature model into a UML model.
- b) Transform each RootFeature into a root Package, which includes a class with the same name as the feature.
- c) Transform each mandatory FeatureGroup (i.e., with minimum cardinality greater than zero, as in example 5 of Table 1) into a super-class of the set of classes generated by its owned GroupedFeature instances and generate an association with the class generated from the owner Feature.
- d) Transform each optional FeatureGroup (i.e., with minimum cardinality equal to zero, as in example 4) into a package merged with the package associated with the parent feature. Then, inside this package, create a super-class of the set of classes generated by its owned GroupedFeature instances and an association with a duplicate of the class generated from the owner Feature.
- e) Transform each mandatory GroupedFeature into a class that specializes the class generated by the owner FeatureGroup (*CreditCard* in example 5).
- f) Transform each optional GroupedFeature into a package merged with the package associated with the parent feature. Then, inside the new package, create a class that specializes a duplicate of the class generated by the owner FeatureGroup (*DebitCard* in example 5).
- g) Transform each mandatory SolitaryFeature typed as FEATURE into a class and associate it with the class generated from the owner feature (*TaxationOp* in example 3).
- h) Transform each mandatory SolitaryFeature (all the subtypes except FEATURE) into a typed attribute and assign this attribute to the class generated from the owner feature (*Login* in example 1).
- i) Transform each optional SolitaryFeature (all the subtypes) into a package merged with the package associated with the parent feature. Then:
 - a. Inside the new package, transform the SolitaryFeature typed as FEATURE into a class and, additionally, associate this class with a duplicate of the class generated from the owner feature with (*ShippingOptions* in example 3).
 - b. Inside the new package, transform the SolitaryFeature with type other than FEATURE into an attribute and assign this attribute to a duplicate of the class generated from the owner feature (*ShippingAdresss* in example 2).

The strategy is based on the three subtypes of *Feature*. The root of every tree in a feature model (*RootFeature*) is transformed into a base package (and an initial class, which will generally be discarded) and a recursive transformation of SolitaryFeatures and FeatureGroups linked to every feature is carried out. The presence of a group implies a class associated to the parent feature that is specialized in several subtypes (one per alternative feature). Previously, a new merging package is created if the feature is optional. A similar set of transformations has been defined in parallel from feature patterns to use case diagrams. The complete definition of both sets of transformations, using QVT [26], can be found in [22]. The package content must be revised and completed, but the package structure itself can be used afterwards to automatically derive the

product model by selecting the desired features. This possibility compensates for the overcharge of complexity that the traceability management and the extensive use of packages in the architectural models entails.

Concerning the implementation details of the transformations, a partial implementation, using XML style sheets, was given in [21]. We used a combination of available tools, based on the Eclipse platform (*fmp* plug-in [11]) and Microsoft proprietary tools. The connection between them is achieved using intermediate XML files that can be automatically generated. The C# language and the Microsoft .NET platform have been selected because of the direct support of the package merge mechanism by means of partial classes. In [21] these experiences are described. The developed product line domains include Web and mobile based families of applications.

Work in progress includes the development, with the Microsoft DSL tools, of a specific domain language, functionally equivalent to the *fmp* eclipse plug-in, to integrate all the steps of the process, from goals analysis to application package configuration inside the Microsoft Visual Studio platform.

4 Related Work

Schobbens *et al.* [27] have reviewed the different variants of feature diagrams, clarifying the differences and establishing generic formal semantics. The influence of non-functional requirements preferences in variant selection has been faced by several methods. The original FODA proposal uses the feature models to represent all the types of variability, functional and non-functional [18]. Jarzabek *et al.* address the non-functional requirements and feature relationships in the PL context [15]. They extend FODA with concepts of goal-oriented analysis. The proposed framework allows developers to record design rationale in the form of interdependencies between variant features and soft-goals, with both models at the same level. Finally Yu *et al.* present a model-driven extension to their Early Requirements Engineering tool (OpenOME) in [31] that generates an initial feature model from stakeholder goals.

Also, the work devoted to relating feature constructions and architectural designs is abundant. Recent proposals express variability with UML models by modifying or annotating these models. Structural, functional or dynamic models have been used. Some authors have proposed explicitly representing the functional variation points by adding annotations or changing the essence of the use case diagrams. For example, Von der Maßen *et al.* [30] propose using new relationships ("option" and "alternative") and the consequent extension of the UML meta-model. John and Muthig [16] suggest the application of use case templates to represent the variability in PLs, using stereotypes. Concerning structural models, either the mechanisms of UML are used directly (through the specialization relationship, the association multiplicity, etc.), as in the case of Jacobson [14]; or the models are explicitly annotated using stereotypes. The work of Goma [10] is an example of this approach, since it uses the stereotypes <<kernel>>, <<optional>> and <<variant>> (corresponding to obligatory, optional, and variant classes). Similarly, Clauß proposes a set of stereotypes to express the variability in architecture models [4].

In the Czarnecki meta-model approach [9], the features are typed (including FEATURE type as default type and String, Integer, etc.). Additionally, we use other elemental types such as Date, Time, Money, directly translatable to conventional programming languages.

The mapping between requirements and design has always been considered complex for several reasons (flexibility and adaptability of the PL, technology options, availability of resources, etc.) [28]. The classical works of Kang and Lee [17, 18, 23], Czarnecky [7], Griss *et al.* [12], or Bosh [2] among others, have enabled the identification of a set of feature patterns that can potentially populate the intended catalog. Sochos *et al.* provide an analysis of the PL methods and propose strengthening the mapping between requirements and architecture by modifying the feature models [28]. The disadvantage is the introduction of implementation characteristics in the requirements models. Bragança *et al.* [3] propose the reverse transformation, obtaining features from use case

models. Their solution requires the UML meta-model extension and the use of annotations. Another solution, proposed by Czarnecki [6], consists of annotating the UML models with presence conditions, so that each optional feature is reflected in one or, more realistically, several parts of a diagram (which may be a class, an association, an attribute, etc. or a combination of elements). Although the class diagrams are the most used, the technique can be applied to any UML model, in particular the sequence or activity diagrams. Some tools are provided, such as an Eclipse plug-in for the definition and configuration of the feature model and an auxiliary tool to show the presence condition in UML models. We think that our package approach is a better solution as it uses conventional UML CASE tools.

5 Conclusions and future work

In this article, the possibilities provided by the combination of different modeling paradigms to represent and configure variability in a product line are discussed. The main contribution is the identification of patterns in feature models and their mapping into the corresponding architectural structures. The organization of the product line in packages that represent the common and optional parts is taken into account as an integral part of the transformations.

The feature patterns catalog enables the automated creation of traceability links between the product line feature and the architectural models. A set of transformations based on QVT are defined to obtain the UML behavioral and structural models, including the package structures. Work in progress includes the development of a specific domain language (based on the feature meta-model) that will connect all the phases of the process, from goal analysis to the application package configuration in the same tool, using these traceability links.

As future work, an advanced vision is based more strictly on the MDE paradigm, automating most of the phases of product line development. Firstly, the set of UML structural and behavior models are obtained using the feature pattern transformations (manual completion of these models will always be required). Then, the goal based configuration process yields a subset of packages that will be merged into a (platform independent) combined model using existing MDE tools. The resulting PIM will be used as input to code generator tools. These tools are precisely intended to generate the platform specific models and the final code. We are evaluating some of the best known tools in order to assess the practical possibilities of the product line and MDE paradigms alliance.

Acknowledgements

This work has been supported by the Junta de Castilla y León (project VA018A07).

References

1. Antkiewicz, M., Czarnecki, K.: Feature modeling plugin for Eclipse. In OOPSLA'04 Eclipse technology exchange workshop (2004)
2. Bosch, J.: "Design & Use of Software Architectures. Adopting and Evolving a Product-Line Approach". Addison-Wesley (2000)
3. Bragança, A., Machado, R. J.: "Automating Mappings between Use Case Diagrams and Feature Models for Software Product Lines", Proceedings of SPLC 2007, pp 3-12, Kyoto, Japan, IEEE CS Press (2007)
4. Clauß, M.: Generic modeling using Uml extensions for variability. In Workshop on Domain Specific Visual Languages at OOPSLA 2001 (2001)
5. Clements, P. C., Northrop, L.: "Software product lines: Practices and Patterns". SEI Series in Software Engineering, Addison-Wesley (2001)

6. Czarnecki, K., Antkiewicz, M.: Mapping features to models: a template approach based on superimposed variants, In proceedings of GPCE'05, LNCS 3676, Springer, pp. 422-437 (2005)
7. Czarnecki, K., Eisenecker, U. W.: "Generative Programming: Methods, Tools, and Applications", Addison-Wesley (2000)
8. Czarnecki, K., Helsen, S.: "Classification of Model Transformation Approaches". OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture (2003)
9. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged Configuration Through Specialization and Multi-Level Configuration of Feature Models. *Software Process Improvement and Practice*, 10(2), pp. 143 – 169 (2005)
10. Gomma, H.: Object Oriented Analysis and Modeling for Families of Systems with UML. In W. B. Frakes, editor, *IEEE International Conference for Software Reuse (ICSR6)*, pages 89–99, (2000)
11. González-Baixauli, B., Leite J.C.S.P., Mylopoulos, J. "Visual Variability Analysis with Goal Models". *Proc. of the RE'2004*. Kyoto, Japan. IEEE Computer Society, 2004. pp: 198-207 (2004)
12. Griss, M.L., Favaro, J., d'Alessandro, M., "Integrating feature modeling with the RSEB", *Proceedings of the Fifth International Conference on Software Reuse*, p.76-85. (1998)
13. Halmans, G., and Pohl, K., "Communicating the Variability of a Software-Product Family to Customers". *Journal of Software and Systems Modeling* 2, 1, 15—36 (2003)
14. Jacobson I., Griss M. and Jonsson P. "Software Reuse. Architecture, Process and Organization for Business Success". ACM Press. Addison Wesley Longman. (1997)
15. Jarzabek, S.; Yang, B.; Yoeun, S., "Addressing quality attributes in domain analysis for product lines," *Software, IEE Proceedings -* , vol.153, no.2, pp. 61-73, (2006)
16. John, I., Muthig, D.: Tailoring Use Cases for product line Modeling. *Proceedings of the International Workshop on Requirements Engineering for product lines 2002 (REPL'02)*. Technical Report: ALR-2002-033, AVAYA labs, (2002)
17. Kang, K. C., Kim, S., Lee, J. y Kim, K. "FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures". *Annals of Software Engineering*, 5:143-168 (1998)
18. Kang, K. C., Cohen, S., Hess, J., Nowak, W., Peterson, S.: "Feature-Oriented Domain Analysis (FODA) Feasibility Study". Technical Report, CMU/SEI-90-TR-21, Software Engineering Institute (Carnegie Mellon), Pittsburgh, PA 15213 (1990)
19. Kazman, R., Klein, M., Clements, P.: 'ATAM: method for architecture evaluation', Technical Report CMU/SEI-2000-TR-004, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA (2000)
20. Laguna, M.A., González-Baixauli, B.: Product Line Requirements: Multi-Paradigm Variability Models. *Proceedings of the 11th Workshop on Requirements Engineering WER 2008*, Barcelona, Spain (2008)
21. Laguna, M.A., González-Baixauli, B., Marqués, J.M.: Seamless Development of Software Product Lines: Feature Models to UML Traceability. *Sixth International Conference on Generative Programming and Component Engineering (GPCE 07)*. Salzburg, Austria (2007)
22. Laguna, M.A., González-Baixauli, B., Marqués, J.M.: Feature Patterns and Multi-Paradigm Variability Models, *GIRO Technical Report 2008/01*, University of Valladolid (2008). Available at <http://www.giro.infor.uva.es/Publications/>
23. Lau, S.: "Domain Analysis of E-Commerce Systems Using Feature-Based Model Templates", MASC Thesis, ECE Department, University of Waterloo, Canada, 2006.
24. Lee, K., Kang, K. C., Chae, W., Choi, B. W.: "Feature-Based Approach to Object-Oriented Engineering of Applications for Reuse". *Software: Practice and Experience*, 30(9):1025-1046. 2000.
25. Object Management Group: "MDA Guide Version 1.0" (2003)
26. 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)
27. Schobbens, P., Heymans, P., Trigaux, J., and Bontemps, Y. Generic semantics of feature diagrams. *Comput. Netw.* 51, 2, 456-479 (2007)
28. Sochos, P., Philippow, I., Riebish, M.: Feature-oriented development of software product lines: mapping feature models to the architecture. Springer, LNCS 3263, pp. 138-152 (2004)
29. van Deursen, A., Klint P.: "Domain-Specific Language Design Requires Feature Descriptions". *Journal of Computing and Information Technology* 10(1):1-17 (2002)
30. von der Massen, T., Lichter, H.: "RequiLine: A Requirements Engineering Tool for Software product lines". In *Software Product-Family Engineering, PFE 2003*, Siena, Italy, LNCS 3014 pp 168-180 (2003)
31. Yu, Y., Lapouchnian, A., Liaskos, S., Mylopoulos, J., Leite., J.C.S.P.: From goals to high-variability software design, pp. 1-16, In: *17th International Symposium on Methodologies for Intelligent Systems (ISMIS'08)* (2008)