Product Line Requirements: Multi-Paradigm Variability Models

Miguel A. Laguna
University of Valladolid
mlaguna@infor.uva.es

Bruno González-Baixauli
University of Valladolid
bbaixauli@infor.uva.es

Abstract

One of the most important issues in the development of software product lines is the elicitation, management, and representation of the variability. In this context, one of the most used instruments is the feature model. But a feature model (due to the open definition of feature) usually contains an amalgamation of various different variability aspects as structural, behavioral, non-functional, or platform variability. We propose to separate these variability aspects of the product line, using other models as goals or UML diagrams but keeping features as the core model. The second part of the article explores the possibilities of identifying mappings between the feature models and the correspondent architectural counterparts. With these mappings, the automated creation of traceability links between the product line models is possible and hence the productivity in the development process of the product line will be enhanced. This approach also simplifies the separation in several development stages, using the appropriate paradigms as goals, features, package models, platforms…

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], [4]. A PL itself is a set of reusable assets, where three abstraction levels can be clearly identified (requirements, design and implementation). In the requirements 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 domain architecture of the product line. Later, the architecture of a single product must be derived from this domain architecture. In this process, the customer functional and non-functional requirements for the product are used for choosing among alternative features. This activity can be seen as a transformation process where a set of decisions at the requirements level generates the product feature model and, consequently, via traceability, the architecture of the product as proposed in [7].

Therefore, one of the most critical points is the elicitation and analysis of variability in the product line requirements. In addition to the information that expresses the requirements themselves it is important to know the variability of these requirements, and the dependencies between them. In this context, feature models are the basic instrument to analyze and configure the variability and commonality of the software family. But although its effectiveness has been proven in many projects, these models are oriented to the solution (what characteristics should have the products) more than to the requirements (what must do the products). On the other hand, non-functional aspects are very difficult to express as a feature because of their fuzzy nature. Consequently, the use of the feature diagrams as a monolithic tool over-simplifies and limits the potential of the technique.

We propose to use additionally techniques from Goal Oriented Requirements Engineering (GORE) and some of the well known UML diagrams. This proposal assumes that more than a unique view is needed to express the diverse variability aspects of a PL. In this context, the relationships and traceability links between models are critical. The second part of the article discusses some of the most interesting of these relationships.

The rest of the paper is as follows: The next Section discusses the separation of the variability model in several views, enabling to work in several abstraction levels. Section 3 analyses related work and identifies mappings between features and architectural UML models. Section 4 concludes the paper and proposes additional work.

2 Multi-paradigm PL Requirements

Originally the Feature Oriented Domain Analysis (FODA)[15] 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 (Figure 1). As long as its parent feature is included: a Mandatory feature must be included in every member of a product line; an Optional feature may be included; exactly one feature from a set of Alternative features must be included; and any non-empty subset from a set of Or features should be included. Several improvements have been proposed (see [19] for a comparison). In particular a (min:max) cardinality can be used for both ends of the parent/child relations. As a natural extension, the general cardinality (m, n) can indicate mixed type of feature decompositions.

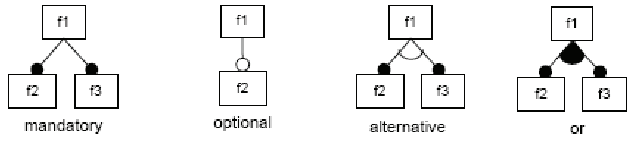


Figure 1. Basic FODA constructions

Other extensions [1] add an atomic kind of feature for some of the leaves of the feature tree with attribute value (String, Integer, etc.). This aspect is interesting if we seek to transform feature diagrams into architectural models. But the original idea of feature models tries to cover simultaneously different variability models: it represents structural variability, behavioral variability, non-functional variability, etc. The differentiation of these different aspects, separating them in different models, can improve the development of the PL. We propose to use the feature model as the central piece of the puzzle that connect the rest of the models, better at expressing different aspects of variability. Features are also the best tool for the PL configuration process.

To start the analysis of the different facets of variability that feature amalgamates we look at the classification of features in literature. FODA [15] classifies the commonality and variability aspects in:

- The capabilities of applications in a domain from the perspective of the user. They are user visible characteristics that can be identified as services, operations, or non-functional characteristics.
- The operating environments (hardware and software platforms, including operating systems) in which applications are used and operated.
- The application domain technology based on which requirements decisions are made (including laws, standardization, business rules).
- The implementation techniques (algorithms or data structures).

Jarzabek *et al* [14] reorganizes the PL requirements in features and quality attributes or non-functional requirements (NFR). The former can be categorized into behavioral requirements that represent functionality or services, and design decisions that describe how the system should behave in particular situations.

These different types and sub-types of features can be studied separately. Capabilities category of features includes very different aspects: the structural aspect that can be represented by classical domain models (UML class diagram); the behavior facets that can be expressed by UML use case models; and the objectives or NFR that can be analyzed better with goal models.

The Goal Oriented Requirements Engineering proposes an explicit modeling of the intentionality of the system (the “whys”). Intentionality has been widely recognized as an important point of the system, but it is not usually modeled. The main advantages of the goal-oriented approaches are that they can be used to study alternatives in software requirements (it uses AND/OR models for the different alternatives) and that they can relate functional and NFR. A goal is an objective that the system under consideration should achieve [21]. There are two types of goals: (hard) goals and *softgoals*: goal satisfaction can be established through verification techniques, but *softgoal* satisfaction cannot be established in a clear-cut sense (it is usually used to model non-functional characteristics of the system) [3]. The dependence between goals and *softgoals*, and consequently between functional and NFR, can be established, by example with the relationships defined by the NFR Framework [3] that model positive and negative correlations between them.

The high level features usually represent the aims of the product line and can be represent better as (hard) goals models. Since *softgoals* are specifically used to introduce the non-functional aspects, they are a good choice to represent the more abstract features. Also, since they are more related with the problem, and therefore to the customers, they can be used to select a particular product, introducing a rationale basis during the product derivation process. A tool that can evaluate these goals and *softgoals* models automatically with respect to the customer preferences has been built to support our approach [10]. Of course, a set of traceability links between *softgoals,* goals, features, and UML models must be carefully established.

In short, we propose to limit the use of feature models to express structural and functionally variability, using the feature model to connect the rest of the techniques and to allow the derivation from more abstract (goal models) to more refined models (architectural models). An obvious consequence is that the feature category is valuable information that can be added to the feature diagrams. In fact, nothing prevents us of assigning several categories to the same feature, associating it with behavioral o structural models (e.g. a feature group for payment types results in a specialization class structure and a use case diagram with <<extend>> relationships).

The UML models are conventional diagrams that are organized in packages. In [16] we proposed a technique where the common features are organized in a base package, and each optional feature in a package which includes the set of UML diagrams that are the solution that achieve this feature. The packages are related using the UML package merge mechanism.

The platform variants must be considered in a second stage, as (at the requirements level) most of the variation points are independent of the operating environment. The main contribution the Model Driven Engineering (MDE) paradigm is the separation of the PIM (platform independent model) from the PSM (platform specific model). Using this approach the operating environment category of features can be analyzed in second term, after the capabilities features have been considered.

Finally the last two groups of features (application domain and implementation techniques) are too specific to introduce significant differences in the general variability analysis. Algorithm implementation details or legal constraints are important information about the common aspects of the product line but not in general from the point of view of the variability analysis.

Figure 2 summarize our proposal as combination of several paradigms:

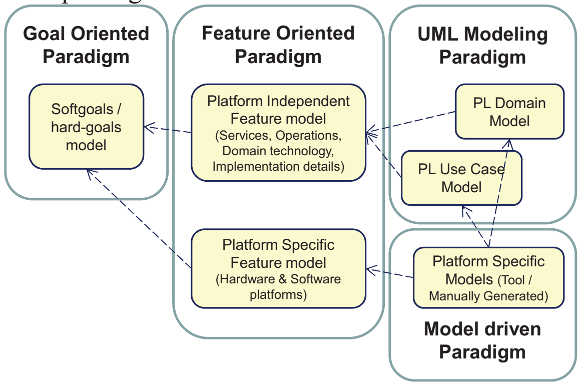

Goal Oriented Paradigm
Softgoals / hard-goals model
Feature Oriented Paradigm
Platform Independent Feature model (Services, Operations, Domain technology, Implementation details)
Platform Specific Feature model (Hardware & Software platforms)
UML Modeling Paradigm
PL Domain Model
PL Use Case Model
Platform Specific Models (Tool / Manually Generated)
Model driven Paradigm

Figure 2. Combination of paradigms and variability

- The goal models represent the intention of the product line, i.e. the high level objectives the application must solve, and the non-functional characteristics.
- The feature model represents the end-user functional requirements, connected with the hard goals of the goal models.
- The UML models organize the architecture of the PL, connected with the feature model.
- The features that configure the operating environment must be considered in a later stage, as we adopt the MDE paradigm of separation of the PIM/PSM models.
- The information about the details of frameworks, platforms, etc. is kept apart from the platform independent models.

Once the product line is developed, the next step is product derivation. Figure 3 shows the schematic view of the process of configuration of an application using our approach: first, using the tool described in [10] we find the optimal combination of goals and *softgoals* for the satisfaction of the customer needs. This combination originates the configuration of the feature application model and the package configuration for the concrete application with the basic architecture. These steps can be totally automated. From here, two alternative ways are open: manually complete the application or use a MDE code generation tool. The experiences so far consist of manually adding the user interface and persistence details to the UML package models. The platform specific models are based in Microsoft .NET as this platform allows implementing directly the concept of package merge using C# partial classes. This manual approach has been successfully applied to the development of product lines in the Web and mobile applications domains. An alternative under study is to use code generation tools as AndroMDA or OpenMDX.

But the productivity in PL development demands to automate the construction and configuration of the diverse PL models, as the MDE paradigm advocates. The transformation from goal to feature model has been treated by Yu *et al.* in [22] where they use a catalog of goal patterns and maps them to their corresponding feature constructions. The next section deals with the analogous transformation of feature models (functional and structural features) into architecture level models.

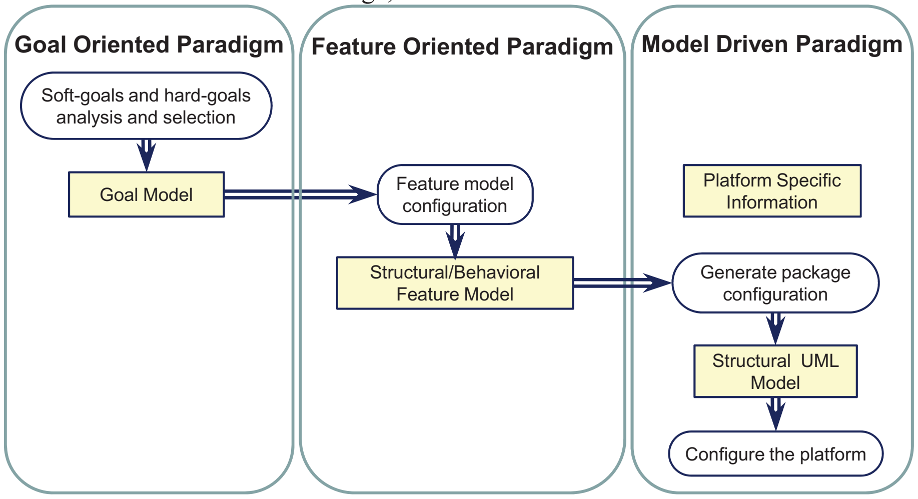

Goal Oriented Paradigm
Soft-goals and hard-goals analysis and selection
Goal Model
Feature Oriented Paradigm
Feature model configuration
Structural/Behavioral Feature Model
Model Driven Paradigm
Platform Specific Information
Generate package configuration
Structural UML Model
Configure the platform

Figure 3. Combination of paradigms in the application derivation process of a product line

3 Features to UML Mapping

In this section we present a catalog of commonly used derivations of feature to UML models. A revision of the literature has revealed that it is naive to pretend a simple and univocal transformation from feature models to UML diagrams. Therefore, we have adopted a pragmatic and multi-view approach: separate the different categories of features in a variability model and treat each of these categories in a different way. Sochos *et al.* [20] have reviewed recently the approaches apart from proposing a new one. An analysis of previous work ([15], [13], [18], [6], [11], [2]) have allowed a set of possible mappings between feature and UML models to be identified. To illustrate them we use a selection of examples extracted from a large case study about e-commerce that uses feature models, class models and activity diagrams [17].

We differentiate two kinds of transformations: structural information mapped to class diagrams, and behavioral features mapped to use case diagrams. We can annotate the features as structural or behavioral oriented (or both). In this article we focus on structural mappings.

Our proposal is based in the Czarnecki *et al.* metamodel [8], where as we discuss before, the features can be typed with data types (String, Integer, etc). In this context we have identified several mappings:

- The presence of a mandatory feature of default 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 default 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 of 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 a class that represents the parent feature.

The most common architectural equivalence of the grouped features (alternative and OR groups) is based in inheritance. Really, a combination of generalization and composition relationships is needed to differentiate alternative from OR structures (Figure 4).

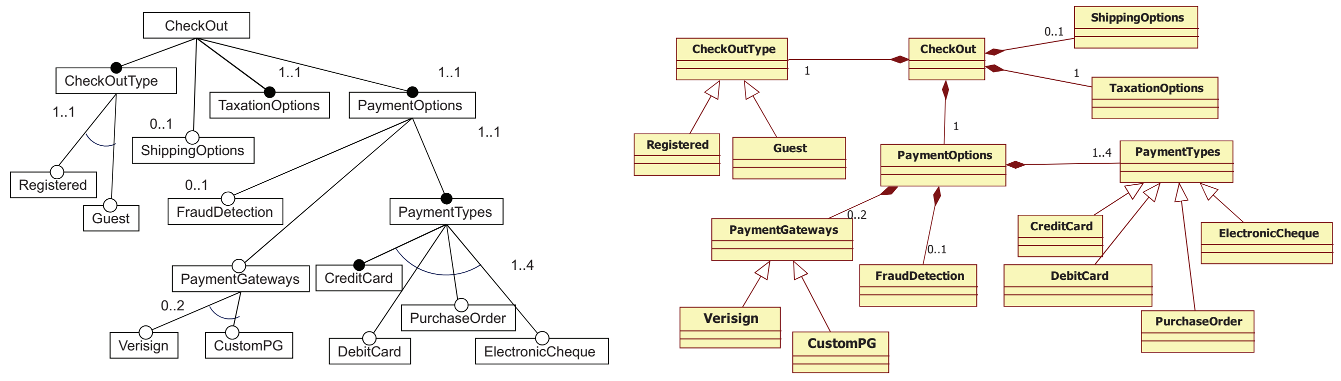
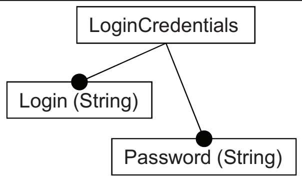
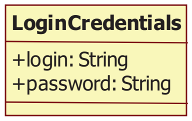
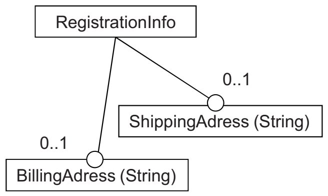
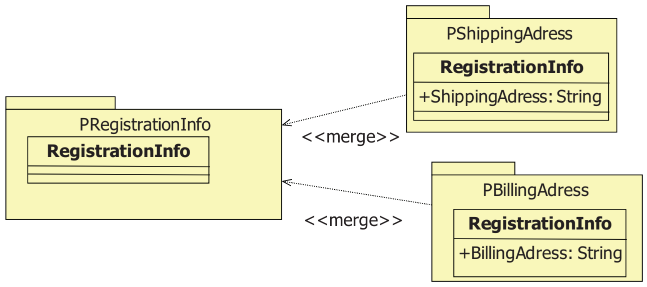
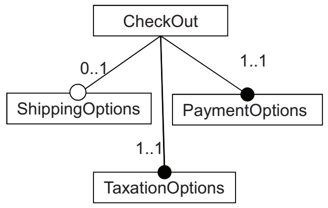
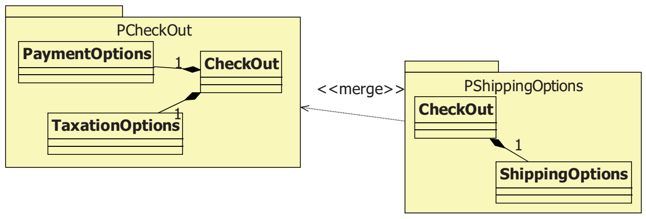
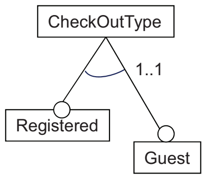
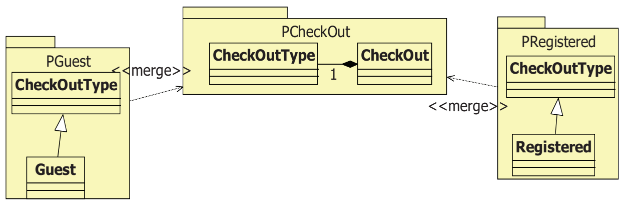
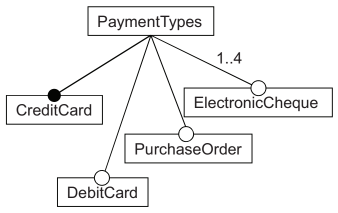
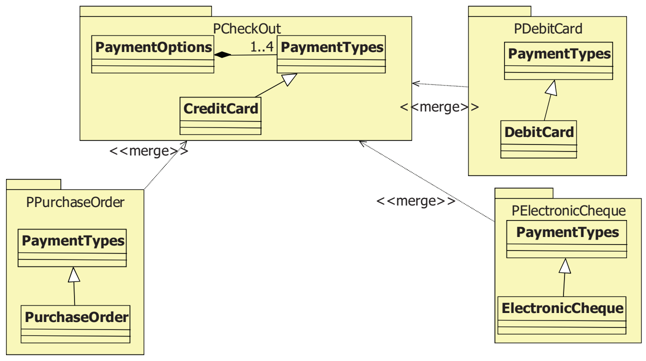


Figure 4. Structural feature model fragment with examples of alternative and OR feature constructions

The problem is that this approach does not have into account the difference between PL variability and the possible variability of the products. Most authors use stereotypes, annotating some classes as variants or optional elements [9], [12], and others use specialized superimposition UML diagrams [5]. We have discussed in previous works the disadvantages of these approaches and proposed to use a standard element of UML 2: “package merge mechanism” that basically consists of adding details to the models in an incremental way. According to the specification of UML 2, <<merge>> is defined as a relationship between two packages that indicates that the contents of both are combined, allowing to extend the modeled concept incrementally in each separate package. Selecting the desired packages, it is possible to obtain a tailored definition from all the possible ones. This mechanism allows a clear traceability between feature and UML models to be established.

The Table 1 combines the literature view with the package merge based interpretations. The classical version of Figure 4 uses only the multiplicity of the attributes and associations to represent optional features and is more compact. But the proposed representation is preferable as removes any ambiguity and is directly mapped to code. The apparent complexity of the package model reflects the real complexity of the product line itself and it is easily handled by the current CASE and IDE tools.

Table 1. The basic structural features and their translation to packaged class diagrams

Feature Construction	Package / Class Structure	Explanation
		Simple type features are mapped to class attributes. Multiplicity is 1..1 since they are mandatory.
 RegistrationInfo 0..1 ShippingAdress (String) 0..1 BillingAdress (String)	 PShippingAdress RegistrationInfo +ShippingAdress: String PRegistrationInfo RegistrationInfo <<merge>> PBillingAdress RegistrationInfo +BillingAdress: String <<merge>>	Simple type features are mapped to class attributes. To separate PL and product variability, two different packages with <<merge>> relationship are created (for each optional feature).
 CheckOut 0..1 1..1 ShippingOptions PaymentOptions 1..1 TaxationOptions	 PCheckOut PaymentOptions 1 CheckOut 1 TaxationOptions <<merge>> PShippingOptions CheckOut 1 ShippingOptions	Complex type features (FEATURE) are mapped to classes. Mandatory features in main package and optional in new package with <<merge>> relationship.
	 PCheckOut CheckOutType 1 CheckOut PGuest <<merge>> CheckOutType Guest PRegistered CheckOutType <<merge>> Registered	Complex type features (FEATURE) are mapped to classes. Alternative features are mapped to subtypes in new packages.
 PaymentTypes 1..4 CreditCard ElectronicCheque PurchaseOrder DebitCard	 PCheckOut PaymentOptions 1..4 PaymentTypes CreditCard <<merge>> PDebitCard PaymentTypes DebitCard <<merge>> <<merge>> PPurchaseOrder PaymentTypes PurchaseOrder PElectronicCheque PaymentTypes ElectronicCheque	Complex type features (FEATURE) are mapped to classes. Common feature is on main package, and alternative as different packages (constraints are implicit in feature model).

4 Conclusions and future work

In this article, the possibilities provided by the combination of diverse modeling paradigms to represent and configure variability in a product line are discussed. The use of specialized techniques have shown better results for expressing different aspects of the requirement variability, while the feature model continues being the central piece of the puzzle.

The second contribution of the article is the identification of distinctive structures in the feature models and the mapping of these to the correspondent architectural diagrams. The feature mapping catalog allows the automated creation of traceability links between the product line feature and the architectural models, and consequently the productivity in product line development is improved. The final conclusion is positive as the combination of paradigms and the mapping catalog makes more straightforward the development process of the product line.

As future work, we foresee the automation of the product line development. First, the set of UML domain and behavior models are obtained (manual completion of these models will always be required). Then, the goal based configuration process yields a subset of packages that will be merged at conceptual level in a monolithic model (using existing MDE tools). The resulting (platform independent) model 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 generative tools in order to assess the practical possibilities of this product line and MDE alliance.

Acknowledgements

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

References

[1] M. Antkiewicz, and K. Czarnecki, "Feature modeling plugin for Eclipse". *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] Chung, L., Nixon, B., Yu, E. and Mylopoulos, J. *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers 2000.

[4] Clements, Paul C. and Northrop, L. *Software product lines: Practices and Patterns.* SEI Series in Software Engineering, Addison-Wesley. 2001.

[5] K. Czarnecki, and M. Antkiewicz, "Mapping features to models: a template approach based on superimposed variants", *GPCE'05, LNCS 3676*, Springer, pp. 422-437.

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

[7] K. Czarnecki and S. Helsen. "Classification of Model Transformation Approaches". *OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[8] K. Czarnecki, S. Helsen, and U. Eisenecker, "Formalizing cardinality-based feature models and their specialization", *Software Process Improvement and Practice*, Vol. 10, No. 1, 2005, pp.7-29.

[9] H. Gomaa. "Object Oriented Analysis and Modeling for Families of Systems with UML". *IEEE International Conference for Software Reuse (ICSR6)*, 2000, pp. 89–99.

[10] B. González-Baixauli, J.C.S.P. Leite, and J. Mylopoulos. *"Visual Variability Analysis with Goal Models". RE'2004*. Kyoto, Japan. IEEE Computer Society, 2004. pp. 198-207.

[11] M.L. Griss, J. Favaro, and M. d'Alessandro, "Integrating feature modeling with the RSEB", *Fifth International Conference on Software Reuse*, 1998, pp.76-85.

[12] G. Halmans, and K. Pohl, "Communicating the Variability of a Software-Product Family to Customers". *Journal of Software and Systems Modeling 2, 1* 2003, 15--36.

[13] Jacobson I., Griss M. and Jonsson P. *Software Reuse. Architecture, Process and Organization for Business Success*. ACM Press. Addison Wesley Longman. 1997.

[14] S. Jarzabek, B. Yang, and S. Yoeun, "Addressing quality attributes in domain analysis for product lines," *Software, IEE Proceedings* - , vol.153, no.2, 2006, pp. 61-73

[15] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. "Feature-Oriented Domain Analysis (FODA) Feasibility Study". *Technical Report, CMU/SEI-90-TR-21*, Software Engineering Institute, Pittsburgh, PA 15213

[16] M.A. Laguna, B. González-Baixauli, and J.M. Marqués, "Seamless Development of Software Product Lines: Feature Models to UML Traceability". *GPCE 07*. Salzburg, Austria 2007

[17] S. Lau, "Domain Analysis of E-Commerce Systems Using Feature-Based Model Templates", *MASc Thesis*, ECE Department, University of Waterloo, Canada, 2006.

[18] K. Lee, K.C. Kang, W. Chae, and B.W. Choi, "Feature-Based Approach to Object-Oriented Engineering of Applications for Reuse", *Software: Practice and Experience, 30(9)*, 2000, pp. 1025-1046.

[19] P.-Y. Schobbens, P. Heymans, and J.C. Trigaux, "Feature diagrams: A survey and a formal semantics". In *RE, 2006* pp. 136–145.

[20] P. Sochos, I. Philippow, and M. Riebish. "Feature-oriented development of software product lines: mapping feature models to the architecture". *Springer, LNCS 3263*, 2004, pp. 138-152.

[21] A. van Lamsweerde, "Goal-Oriented Requirements Engineering: A Guided Tour", *IEEE Symposium Requirements Engineering,* 2001, pp. 249-262

[22] Y. Yu, A. Lapouchnian, S. Liaskos J. Mylopoulos and J.C.S.P. Leite. "From goals to high-variability software design", *17th International Symposium on Methodologies for Intelligent Systems (ISMIS'08)*, 2008, pp. 1-16.