Design of a UML profile for feature diagrams and its tooling implementation

Thibaut Possompès, Christophe Dony, Marianne Huchard, Chouki Tibermacine

Design of a UML profile for feature diagrams and its tooling implementation

Thibaut Possompès[1,2], Christophe Dony[2], Marianne Huchard[2], Hervé Rey[1], Chouki Tibermacine[2], and Xavier Vasques[1]

[1] IBM France – PSSC Montpellier
thibaut.possompes,reyherve,xavier.vasques@fr.ibm.com
[2] LIRMM, CNRS and Université de Montpellier 2
possompes,dony,huchard,chouki.tibermacine@lirmm.fr

Abstract. This paper proposes an instrumented approach to integrate feature diagrams with UML models to be used as part of a general approach for designing software product lines and for product generation. The concrete contribution is a new UML profile based on a meta-model synthesising existing feature diagrams semantics using IBM®Rational Software Architect (RSA) implementation. Our feature diagram meta-model integrates consistency checking rules and focuses on a comprehensive, expressive, but not too complex semantics as it aims at helping various domain specialists to express their knowledge with feature diagrams. It is intended to be used in the context of large, complex and multi-domain projects, and at allowing model transformations to derive products. Our RSA implementation makes possible to link feature diagrams with UML model artefacts. It allows traceability between feature models and other different kinds of models (requirements, class diagrams, sequence or activity diagrams, etc.).

Key words: Feature diagrams, UML profile, software product lines, model driven engineering, profile tooling

1 Introduction

IT projects are closely related to professional domains. However, there exist few techniques to gather, link and manage the knowledge related to each domain. Nowadays projects are particularly confronted to problems where instrumentation of specific business domains is required to enhance their efficiency. Examples of these projects are smart buildings, grids, water management, healthcare or food systems. In our case, this research is done in the context of the RIDER project[3], which brings together several actors by the desire of improving energy

[3] The RIDER project ("Research for IT as a Driver of EneRgy efficiency") is led by a consortium of several companies and research laboratories, including IBM and the LIRMM laboratory, interested in improving building energy efficiency by instrumenting it.

efficiency of buildings. This context requires to be able to link several domains, such as building system components, IT infrastructure, air flow modelling, building thermal management, database models, *etc.* Each domain can be modelled by a different stakeholder, or sometimes being based upon a standard.

Software product line approach is perfectly appropriate to manage the variations that can be found in building instrumentation systems. The solutions presented in this paper will be applied to the RIDER project and benefit of its feedback. Our approach is intended to be used as a part of a general approach for software product lines and for product generation.

There exist several commercial tools such as [3,9,21] that allow linking features to model artefacts extracted from various modelling tools; eclipse plugins [20,1,13,10,27,15,12]; and standalone feature modelling tools [22,24,26]. These tools do not implement the latest feature modelling concepts and are not well integrated in UML modelling tools. There also exist several UML profiles [5,30] and meta-models [29] implementations but they address neither the latest UML specification nor all feature model concepts that could be useful in our context.

This paper is organized as follows. Section 2 presents a state of the art of feature models achieved by classifying concepts into categories. Section 3 presents the feature meta-model and Section 4 explains how the corresponding UML profile has been created and implemented using RSA. Section 5 will sum up what has been presented and suggests some perspectives for further study.

2 Synthesis of Existing Feature Diagram Models

Various feature diagram semantics and implementations have been proposed since the initial one given in *FODA* [16]. In order to ease their analysis, Sections 2.1, 2.2, 2.3 and 2.4 propose a taxonomy based on the following criteria:
1 – Feature definition Features are the main element of feature diagrams. They have been first defined by *FODA* [16].
2 – Feature relationships We identify four criteria to classify existing proposals regarding relations among features.

- *2.1 – Hierarchical relationships* This category presents the semantics of the hierarchical relations between features. Several variants can be found in existing work.
- *2.2 – Feature choice constraint relations* This criterion relates to relationships necessary to guide the user through feature selection. It represents concepts such as feature dependency, or mutual exclusion (conflict).
- *2.3 – Mandatory and optional feature identification* This key concept is used to express, in feature diagrams, the commonalities and the variations that can be found in software product lines.
- *2.4 – Sub-feature selection semantics* Describing the way users can select the features to be implemented in a product. Depending on the choices made to represent this concept, the user will have different degrees of freedom, for example, the number of sub-features that can be chosen and the constraints of sub-features choice.

3 – Feature logical groups Allowing to group arbitrary features by business domains, or abstraction layers.
4 – Product and implementation information To determine what kind of information can help implementing a product from a set of selected features.

2.1 Feature Definitions

Existing work The initial definition of features was introduced by Kang et al. in *FODA* [16] which is extended by the *FORM* [17] method. Features are defined as being essential characteristics of applications, described with domain vocabulary.

FORM introduces four different perspectives to enrich the semantics of feature diagrams: capability, operating environment, domain technology and implementation technique. This point underlines that features are concepts able also to model expert concepts involved in the product. Fey et al. [11] add the *pseudo feature* concept in order to allow specialisation with non exclusive alternatives and to avoid feature redundancy thanks to implicit inheritance. Zhang et al. define features [28,29] as being essentially a cohesive set of individual requirements representing the user-visible capability of a software system. Czarnecki et al. [8,6] consider that features are system properties relevant for some stakeholder, and that any kind of functional or non-functional characteristic of a described system can be represented by a feature. This extends the *intention* definition presented by Zhang et al. [28].

Choices made accordingly to existing work The definition proposed by Czarnecki et al. allows to give the most freedom of expressiveness in feature diagrams. Its combination with *FODA* perspectives, which is necessary to link features addressed to different stakeholders, will allow us to apply feature diagrams to the numerous domains involved in a project. The *property* concept introduced by Fey et al. and the *attribute* concept presented by Czarnecki et al. are semantically very close. We could assess that a property could express the same thing that an attribute, *i.e.*, a measurable characteristic. We will use the *property* concept to describe specific feature typed values (*e.g.*, Integer, *etc.*) and how they can influence each other. We extend this concept to allow the customer to choose the property value. Therefore, we will add the necessary semantics to restrict the possible choices to consistent ones.

2.2 Feature Relationships

Hierarchical Relationships

Existing work Features and sub-features can be bound by either decomposition or specialisation. Table 1 illustrates the different variations on these concepts. *FODA* uses the relationship *consists-of* to represent the decomposition notion. The FODA *consists-of* relation [16,4,25] makes possible to link a set of sub-features related to its parent by either *and-decomposition*, which represents a composition of sub-features where all should be present; or *xor-decomposition*,

Table 1. Hierarchy relationship

	Decomposition	Specialisation	
		Enrichment	Realization
FODA [16]	Consists of		
FORM [18]	Composed of	Generalisation Specialisation	Implemented by
Fey et al. [11]	Refine		Provided-by (realize)
Zhang et al. [29]	Decompose Detail (same picture)	Specialize	
Czarnecki et al. [8]	Relation		

indicating that only one of the sub-features should be present. *FORM* [17,18] introduces the relations *Composed-of*, to describe the constituents of a feature; *Generalization / Specialisation*, to specialize or generalize a feature; and *Implemented-by*, to define how a high-level feature can be implemented by a lower-level feature. Fey et al. [11] use two kinds of hierarchical links between features: *refine*, to detail a given feature at a lower abstraction level; and *provided-by*, to link a pseudo-feature with the features which it realizes. *Pseudo-features* express an abstract functionality, quality or characteristic. Fey et al. make possible to build directed acyclic graphs, which is impossible with *FODA*, by allowing one feature to refine several ones. Zhang et al. [29] identify three different kinds of hierarchy relationships: *decomposition*, to refine a feature into its constituents; *detailization*, to identify a feature attributes; and *specialisation*, to add further details into a feature. Czarnecki et al. [8,6,7] decided not to consider relationships between features in favour of entity-relationship or class diagrams.

Choices made accordingly to existing work The different hierarchy relations can be categorized, as shown in the table 1, with the following concepts: *decomposition*, which consists in *detailing* the sub-features that compose the parent feature; *specialisation*, which encompasses the concepts of *enrichment* for sub-features that add functions to the parent feature, and *Realization* (or *implementation*) that describes how a feature can be implemented.

Feature Choice Constraint Relations

Existing work FODA [16] introduced the concept of *composition rules* to describe how features relate to one another: one feature can *require* another one, or two features can be *mutually exclusive*. Riebisch et al. [23] introduce the *hint* relation in order to help the user to choose pertinent features.

Choices made accordingly to existing work The constraints *require* and *conflict* are necessary to ensure a coherent product generation. Furthermore the *hint* relation is also convenient to make recommendation to the user during the feature selection process.

Mandatory and Optional Features Identification

Existing work In the *FODA* [16] and *FORM* [18] specifications and tools, all features are mandatory by default; optionality is represented as a feature property similar to [23,14,29]. Czarnecki et al. [6] use cardinalities to express this concept. For instance, *(1,1)* cardinality describes a mandatory feature and *(0,1)* describes an optional one. Setting a cardinality superior to 1 specify how often the sub-features of the parent can be duplicated.

Choices made accordingly to existing work The cardinality based relationship introduced by Czarnecki et al. [6] brings a very interesting enhancement to the initial definition, useful to describe more complex products from a product line.

Sub-feature Selection Semantics

Existing work *FODA* [16] and *FORM* [18] methods propose two ways to manage sub-feature selection. A simple link between the parent and its sub-features means that there is no choice constraint; equivalent to an *or* binary choice. A semi circle drawn across the links means that there is an alternative choice; equivalent to the *xor* binary choice. Likewise, Fey et al. [11] express with a simple link that the choice is similar to an *or*, but the alternative choice is expressed with a complete graph of mutually exclusive constraints among all sub-features. Riebisch et al. [23] and Czarnecki et al. [8] use cardinalities to define how many sub-features can be chosen.

Choices made accordingly to existing work The most convenient semantics relies on cardinality based selection semantics. It allows a maximum flexibility to describe how many sub-features can be selected.

Furthermore, we chose to keep the *or*, *and*, and *xor* groups to ease the feature models semantics understanding for non-IT specialists end-users.

2.3 Feature Logical Groups

Existing work The *FORM* [18] method classifies features into four categories called layers, from a functional point of view. They describe capabilities, the operating environment, domain technologies, and implementation techniques. Riebisch et al. [23] use logical groups to represents aspects valuable to the customer and explain that *abstract features* could be used to encapsulate features related to a given concept. Fey et al. [11] present *feature-sets* to group features from an arbitrary point-of-view. Zhang et al. [29] present the *binding time* concept to represent the phases of the software life-cycle in which each feature must be chosen. Czarnecki et al. [6,8] use abstract features to reference other feature diagrams to reuse a set of features. They also introduce different types of features that can be considered as feature groups.

Choices made accordingly to existing work The *binding time* concept can be extended according to the software development process chosen by the final user to allow him to assign features or groups of features to a development phase. It can be modelled by the *feature set* concept presented by Fey et al. The different kinds of features presented by Riebisch et al. [23] could be easily modelled by sub layers. Hence, we choose to keep the layer concept to organize features in logical groups and sub-groups accordingly to the type of information they represent. We propose to enhance Fey et al. *feature-set* concept by adapting Zhang et al. [29] constraints meta-model to describe constraints inside a group of features, and constraints between two groups of features. The feature-sets could also be used along with the Kano method [19] to help the user choosing a set of product features that yield high customer satisfaction. The customer preference categories can be modelled as feature-sets encompassing the corresponding features. We also keep the feature-sets idea to reference sub-parts of a feature diagram. Hence, a feature-set could be a leaf of the diagram that encompasses another feature hierarchy. Staged configuration can be modelled by creating one feature-set for each configuration stage, and associating it with a group of stakeholders that have the same business concern. Czarnecki et al. [6,8] have presented four types of features that have been integrated in our meta-model proposal: *concrete features* can be stored in the implementation layer; *aspectual features* can be stored in a sub layer of implementation layer; *abstract features*, *e.g.* performance requirements, can be represented by feature properties; *grouping features* can be modelled as a feature set.

2.4 Product and Implementation Informations

Existing work Riebisch et al. [23] argue that the feature hierarchy must be organised to make easier the choice of features by using composition relations and require associations. Zhang et al. [29] present a feature attribute for representing the feature *binding-state*. It must be used in a *binding-time* context, *i.e.* when features are implemented in the software product at a given software life-cycle phase. Mathematical relationships have been presented to describe the relative impact of one feature to another.

Choices made accordingly to existing work We will reuse the binding-time concept introduced by Zhang et al. to know when each selected feature must be implemented in the software life-cycle. We will model this information with feature-sets.

3 A Synthesis Meta Model for Feature Diagrams

This section describes a meta model synthesising the different important points that we previously identified. This step is required to facilitate the UML profile creation. This work is based upon and complete the work of Asikainen et al. [2] in view of its usage in a rich industrial project. Furthermore, we describe in the

next section how to integrate this work in the UML standard in order to use it in any UML 2 compliant modelling tool.

3.1 Description

As depicted in Figure 1, a product line contains features, a feature belongs to one product line. A product belongs to one product line and can be composed of as many features as needed. Features associated to a product must be analysed in order to check whether all constraints, like mutual exclusion between features or require relations, are satisfied.

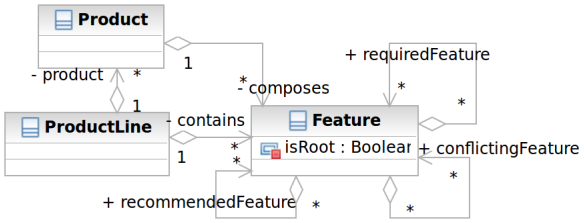

Fig. 1. Product lines relation to features and products

Mutual exclusion and require relations link features regardless of their position in the hierarchy, they are modelled by the *conflictingFeature* and *requiredFeature* relationships. Furthermore, we add the *recommendedFeature* role that advises the user to choose another feature that could be pertinent in the product. These roles are depicted in Figure 1 in reflective relationships on the *Feature* class.

Feature properties (Figure 2) are used to describe either a feature parameter related to its inner requirements (*e.g.* the bandwidth capacity of a network) or a characteristic chosen by the user during the product definition (*e.g.* the frequency of automatic backups of a word processing software).

The *VariabilityKind* type is further described below:

- *fixed*, The property value is fixed throughout all products of the product line.
- *variable*, A property value can change, within a product, depending on other features properties, *e.g.* the text buffer size of a text field in the user interface can vary accordingly to the type of information we want to store (*i.e.*, name or address).
- *family-variable*, The property can vary from product to product accordingly to the selected features, *e.g.* the phone book capacity depends on the presence of internal memory property and the sim card capacity.
- *user-defined*, The property value can be freely chosen in a given product, *e.g.*, the frequency of automated backups in a word processing software.

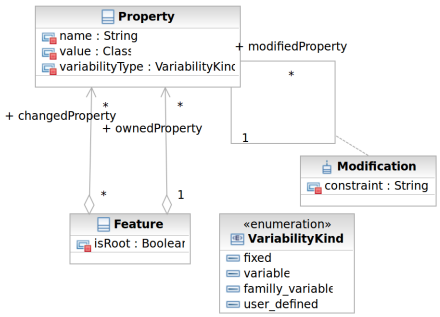

Fig. 2. Feature properties

The hierarchy relationships and sub-feature groups are depicted in Figure 3. The relations between a feature and its sub-features are grouped by the *RelationshipGroup* class that contains the cardinalities necessary to restrict the number of sub-features to choose. Cardinalities can be chosen freely, but some groups have fixed cardinalities: *OrGroup*, *(0,*)*; *AndGroup*, *(*,*)*; and *XorGroup*, *(0,1)*, to clarify the semantics. The *DirectedBinaryRelationship* class represents the kind of association that links the parent feature and a sub-feature. It can be specialised either by *Enrich*, *Implement*, or *Detail* classes.

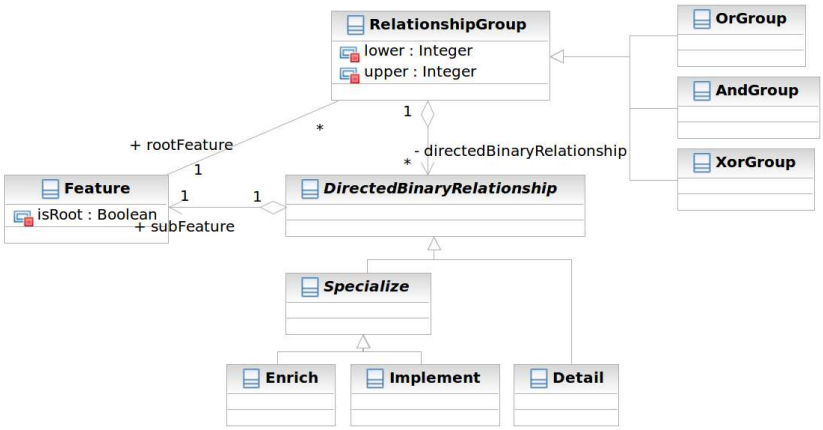

Fig. 3. Groups and hierarchy relationships

Figure 4 shows how layers and feature sets can be associated with the project stakeholders. A stakeholder represents any kind of people (*e.g.* domain experts, IT architects, *etc.*) allowed to choose features. Feature sets and layers are at-

tached to a specific concern related to the project, *e.g.* network architecture, or business requirements. A *Layer* represents a specific view onto the software application, a feature can be in only one layer at a time. A *FeatureSet* inherits from the *Feature* class, and allows grouping features from an arbitrary point of view, *e.g.*, a business domain, or representing the features that must be implemented to fulfil a norm.

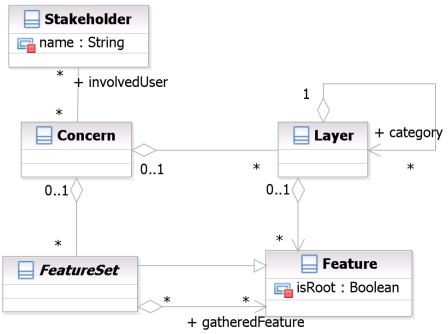

Fig. 4. Stakeholders concerns

Figure 5 depicts how constraints are applied on feature sets, that can be either; *mutex*, when only one feature can be selected in the feature set; *None*, when there is no constraint among features; *All*, when all features or none of them can be selected. The *ConstraintRelation* class describes the relation between two feature sets: one feature set can require another one, two feature sets can mutually require each other, or be mutually exclusive. The *BindingPredicate* is used to represent how a constraint must be applied on each constrained feature-set. The choice of the specialized class must be made according to the kind of feature-set.

In comparison with [2] we add several concepts such as *layers*, *stakeholder concerns*, *feature-sets*, *group constraints*.

4 UML Profile Implementation

The meta model describes the semantics of the elements used in a feature model. The profile reuses the concepts described in the meta model and integrates them in UML thanks to the profile semantics. However, the profile could be implemented in different ways by choosing to extend different meta-classes. We chose the meta-classes that had the closest semantics to our concepts, added the required information with the stereotypes and restricted the initial semantics of meta-classes to what is necessary for our profile with Object Constraint Language (OCL) constraints.

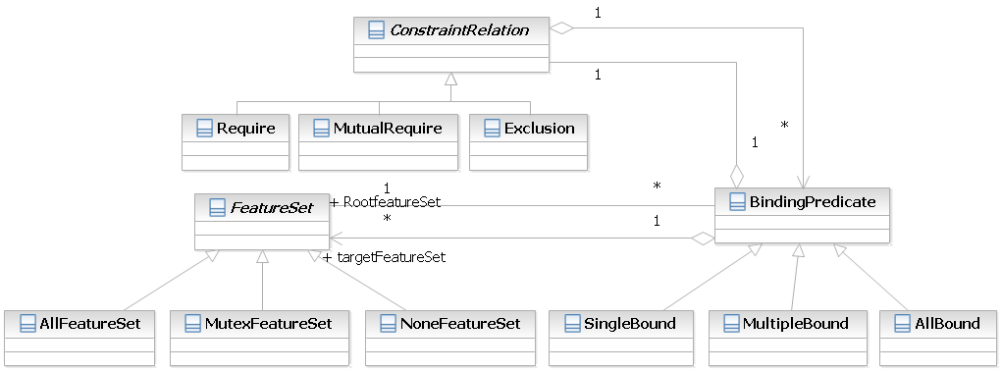

Fig. 5. Constraints on feature sets

Contrary to [5] we choose to base our feature diagram profile on *Components*. It is the UML 2 concept the closest to what we want to express because they are a high level view of software elements. This first choice influences the other meta-classes selection.

Table 2. Stereotypes extensions

Stereotype	**Extended Meta-class**
Feature	Component
Stakeholder	Actor
Concern	Class
ModelRelationship	Dependency
Layer	Package
ProductLine	Component
Product	Component
Property	Port
RelationshipGroup	Port
Modification	Usage
DirectedBinaryRelationship	Association
BindingPredicate	Port
ConstraintRelation	Association

The *port* meta-class allows us to represent the interaction of a feature with other elements. They can be linked to other ports or components. The modification of a property value can be modelled by textual or OCL constraints placed upon the relationship between two properties.

We choose to create a Rational Software Architect plug-in to leverage its UML modeller, modelling editors, views and tools. All models managed are in-

stances of EMF models. Hence using Rational Software Architect allows simplifying tasks like creating a specific plug-in for integrating feature modelling capabilities into standard EMF-based UML models and diagrams. The plug-in is currently used in the RIDER project.

5 Conclusion and Perspectives

We propose a synthesis of existing models, that we enhanced to fit the requirements of the project in which this research is applied, and their profile implementation in UML 2. The models presented in this paper have been instrumented with Rational Software Architect in order to provide a tool able to easily produce feature diagrams using the described profile. This synthesis is achieved by classifying the existing concepts into categories. This approach allows a full integration of feature diagrams into UML models and facilitates future model transformations.

For the time being, we still need to guide users to organise features into layers and sub-layers in order to best integrate them into the software development life-cycle. Hence, the next steps will be to create a framework able to use the full potential of our feature meta-model, and to develop automated model transformation functionalities to automatically generate UML models.

References

1. M. Antkiewicz and K. Czarnecki. FeaturePlugin: feature modeling plug-in for eclipse. In *eclipse '04: Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*, pages 67–72, New York, NY, USA, 2004. ACM.
2. T. Asikainen, T. Mannisto, and T. Soininen. A unified conceptual foundation for feature modelling. In *Proceedings of SPLC '06*, pages 31–40. IEEE Computer Society, 2006.
3. BigLever Gears. http://www.biglever.com/overview/software_product_lines.html.
4. Y. Bontemps, P. Heymans, P. Y. Schobbens, and J. C. Trigaux. Semantics of FODA feature diagrams. In *Proceedings SPLC 2004 Workshop on Software Variability Management for Product Derivation*, pages 48–58, 2004.
5. M. Clauss. *Untersuchung der Modellierung von Variabilität in UML*. Technische Universität Dresden, Diplomarbeit, 2001.
6. K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing cardinality-based feature models and their staged configuration. *University of Waterloo*, 2004.
7. K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration using feature models. *Lecture notes in computer science*, 3154:266–283, 2004.
8. K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process Improvement and Practice*, 10(1):7–29, 2005.
9. M. Eriksson, H. Morast, J. Börstler, and K. Borg. *The PLUSS toolkit: extending telelogic DOORS and IBM-rational rose to support product line use case modeling.* ACM, 2005.
10. Feature-Oriented Software Development Research. http://fosd.de/fide/.

11. D. Fey, R. Fajta, and A. Boros. Feature modeling: A Meta-Model to enhance usability and usefulness. In *Software Product Lines*, pages 198–216. Springer, 2002.
12. fmp2rsm Plug-in . http://gp.uwaterloo.ca/fmp2rsm/index.html.
13. Generative Software Development Lab. http://gsd.uwaterloo.ca.
14. M. L. Griss, J. Favaro, and M. d'Alessandro. Integrating feature modeling with the RSEB. In *In Proceedings of the Fifth International Conference on Software Reuse*, pages 76–85, 1998.
15. Hydra. http://caosd.lcc.uma.es/spl/hydra/index.htm.
16. K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented domain analysis (FODA) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute, Nov. 1990.
17. K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. FORM: a feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5(1):143–168, 1998.
18. K. C. Kang, J. Lee, and P. Donohoe. Feature-Oriented product line engineering. *IEEE Software*, 2002.
19. N. Kano, N. Seraku, F. Takahashi, and S. Tsuji. Attractive quality and must-be quality. *Journal of the Japanese Society for Quality Control*, 1984.
20. ProS Labs - Moskitt Feature Modeler. http://oomethod.dsic.upv.es/labs/index.php.
21. Pure-Systems GmbH. http://www.pure-systems.com/.
22. RequiLine. http://www-lufgi3.informatik.rwth-aachen.de/TOOLS/requiline.
23. M. Riebisch. Towards a more precise definition of feature models. *Modelling Variability for Object-Oriented Product Lines*, pages 64–76, 2003.
24. S2T2 - An SPL of SPL Techniques and Tools. http://download.lero.ie/spl/s2t2/.
25. P. Schobbens, P. Heymans, J. Trigaux, and Y. Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51(2):456–479, Feb. 2007.
26. SPLOT - Software Product Line Online Tools. http://www.splot-research.org/.
27. XFeature. http://www.pnp-software.com/XFeature/.
28. W. Zhang, H. Mei, and H. Zhao. Feature-driven requirement dependency analysis and high-level software design. *Requirements Engineering*, 11(3):205–220, 2006.
29. W. Zhang, H. Zhao, and H. Mei. A propositional logic-based method for verification of feature models. *Lecture Notes in Computer Science*, 3308:115–130, 2004.
30. T. Ziadi, L. Hélouët, and J. M. Jézéquel. Towards a UML profile for software product lines. *Lecture Notes in Computer Science*, pages 129–139, 2004.