Towards a UML Profile for Software Product Lines*

Tewfik Ziadi[1], Loïc Hélouët[1], and Jean-Marc Jézéquel[2]

[1] IRISA-INRIA, Campus de Beaulieu 35042 Rennes Cedex, France
{tziadi,lhelouet}@irisa.fr
[2] IRISA-Rennes1 University, Campus de Beaulieu 35042 Rennes Cedex, France
jezequel@irisa.fr

Abstract. This paper proposes a UML profile for software product lines. This profile includes stereotypes, tagged values, and structural constraints and it makes possible to define PL models with variabilities. Product derivation consists in generating product models from PL models. The derivation should preserve and ensure a set of constraints which are specified using the OCL.

1 Introduction

The Unified Modeling Language (UML) [5] is a standard for object-oriented analysis and design. It defines a set of notations (gathered in diagrams) to describe different aspects of a system: use cases, sequence diagrams, class diagrams, component diagrams and statecharts are examples of these notations. A UML *Profile* contains stereotypes, tagged values and constraints that can be used to extend the UML metamodel.

Software Product Line engineering aims at improving productivity and decrease realization times by gathering the analysis, design and implementation activities of a family of systems. Variabilities are characteristics that may vary from a product to another. The main challenge in the context of software Product Lines (PL) approach is to model and implement these variabilities. Even if the product line approach is a new paradigm, managing variability in software systems is not a new problem and some design and programming techniques allows to handle variability; however outside the Product Line context, variability concerns a single product, i.e variability is inherent part of a single software and is resolved after the product is delivered to customers and loaded into its final execution environment. In the product line context, variability should explicitly be specified and is a part of the product line. Contrarily to the single product variability, PL variability is resolved before the software product is delivered to customers. [1] calls the variability included in a single product "the run time variability", and the PL variability is called "the development time variability". UML includes some techniques such as inheritance, cardinality range, and class template that allow the description of variability in a single product i.e variablity is specified in the product models and resolved at run time. Furthermore, it is interesting to use UML to specify and to model not only one product but a set of products. In this case the UML models should be considered as reference models from which product models can be derived and created. This variability corresponds to the product line variability. In this paper we consider this type of

* This work has been partially supported by the FAMILIES European project. Eureka Σ! 2023 Program, ITEA project ip 02009.

F. van der Linden (Ed.): PFE 2003, LNCS 3014, pp. 129–139, 2004.

variability and we use UML extension mechanisms to specify product line variability in UML class diagrams and sequence diagrams. A set of *stereotypes*, *tagged values* and structural *constraints* are defined and gathered in a UML profile for PL.

The paper is organized as follows: Section 2 presents the profile for PL in terms of stereotypes, tagged values and constraints, Section 3 presents the use of this profile to derive product models from the PL, Section 4 presents related work, and Section 5 concludes this work.

2 A UML Profile for Product Lines

The extensions proposed here for PL are defined on the UML 2.0 [5] and they only concern the UML class diagrams and sequence diagrams. We use an ad - hoc example to illustrate our extensions. The example concerns a digital camera PL. A digital camera comports an interface, a memory, a sensor, a display and may comport a compressor. The main variability in this example concerns the presence of the compressor, the format of images supported by the memory, which can be parameterized and the interface supported. We distinguish three types of interfaces: Interface 1, Interface 2, and Interface 3.

2.1 Extensions for Class Diagrams

UML class diagrams are used to describe the structure of the system in terms of classes and their relationships. In the context of Product Lines, two types of variability are introduced and modeled using stereotypes.

Stereotypes

- **Optionality.** Optionality in PLs means that some features are optional for the PL members. i.e: they can be omitted in some products. The stereotype `<<optional>>` is used to specify optionality in UML class diagrams. The optionality can concern classes, packages, attributes or operations. So The `<<optional>>` stereotype is applied to *Classifier*, *Package* and *Feature* metaclasses.
- **Variation.** We model variation point using UML inheritance and stereotypes: each variation point will be defined by an abstract class and a set of subclasses. The abstract class will be defined with the stereotype `<<variation>>` and each subclass will be stereotyped `<<variant>>`. A specific product can choose one or more variants. These two stereotypes extend the metaclass *Classifier*. The alternative variability especially defined in feature driven approaches is a particular case of our variation variability type where each product should choose one and only one variant. This can be modeled using OCL (Object Constraint Language) [10] as a mutual exclusion constraint between variants. The mutual exclusion constraint will be presented in Section 3.

Constraints. A UML profile also includes constraints that can be used to define structural rules for all models specified by the defined stereotypes. An example of such profile

constraint concerns the stereotype <<variant>>. It specifies that all classes stereotyped <<variant>>, should have one and only one ancestor among all its ancestors stereotyped <<variation>>. This can be formalized using the OCL as follows:

context <<variant>>
inv: self.supertype → select(oclIsKindOf(Variation))→size()=1

Example. Figure 1 shows the class diagram of the camera PL example, the `Compressor` class is defined with the stereotype <<optional>> to indicate that some camera products do not support the compression feature. The camera interface is defined as an abstract class with three concrete subclasses: Interface 1, Interface 2, and Interface 3. A specific product can support one or more interfaces, so the stereotype <<variation>> is added to the abstract class `Interface`. All subclasses of the interface abstract class are defined with the stereotype <<variant>>. Notice that the class diagram of the camera PL includes a class template `Memory` with a parameter that indicates the supported format of images. This type of variability is resolved at run time and all camera products include it.

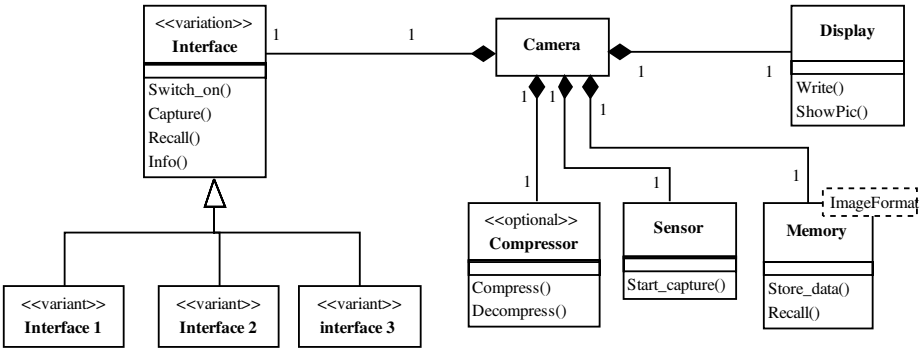

Fig. 1. The class diagram of the Camera PL

2.2 UML Extensions for Sequence Diagrams

In addition to class diagrams, UML includes other diagrams that describe other aspects of systems. Sequence diagrams model the possible interactions in a system. They are generally used to capture the requirements, but can then be used to document a system, or to produce tests. The UML 2.0 [5] makes sequences diagrams very similar to the ITU standard MSC (Message Sequence Chart)[7]. It introduces new mechanisms, especially interaction operators such as alternative, sequence, and loop to design respectively a choice, a sequencing, and a repetition of interactions. [11] proposes three constructs to introduce variability in MSC. In this subsection we formalize this proposition in terms of extensions on the UML 2.0 metamodel for sequence diagrams. Before describing these extensions, we briefly present sequence diagrams as defined in UML 2.0 metamodel.

Sequence Diagrams in UML 2.0. Figure 2 summarizes the UML 2.0 metamodel part that concerns sequence diagrams (interested readers can consult [5] for a complete description of the metamodel). The *Interaction* metaclass refers to the unit of behavior that focuses on the observable exchanges of information between a set of objects in the sequence diagram. The *Lifeline* metaclass refers to the object in the interaction. *InteractionFragment* is a piece of an interaction. The aggregation between the *Interaction* and the *InteractionFragment* specifies composite interaction, in the sense that an interaction can enclose other sub-interactions. The *CombinedFragment* defines a set of interaction operators that can be used to combine a set of *InteractionOperand*. All possible operators are defined in the enumeration *InteractionOperator*. The *EventOccurence* metaclass refers to events that occur on a specific lifeline, these events can be either sending, receiving messages or other kinds. The notation of an interaction in a sequence diagram is a solid-outline rectangle (see Figure 3 for example). The keyword **sd** followed by the name of the interaction in a pentagon in the upper left corner of the rectangle.

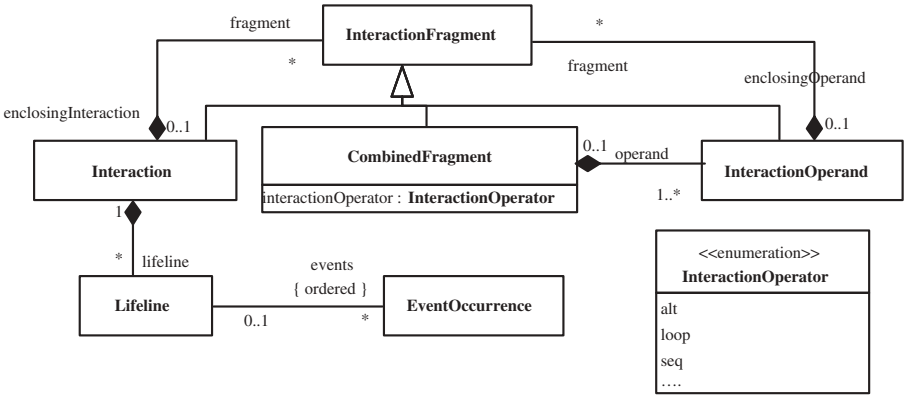

Fig. 2. UML 2.0 metamodel: Interaction part [5]

Stereotypes and Tagged Values. Variability for sequence diagrams is introduced in terms of three constructs: Optionality, Variation and Virtuality, in what follows we formalize these constructs using stereotypes and tagged values on the UML 2.0 metamodel.

- **Optionality.** Optionality proposed for sequence diagrams comports two main aspects: optionality for objects in the interaction, and optionality for interactions themselves. Optionality for object is specified using the stereotype `<<optionalLifeline>>` that extends the *Lifeline* metaclass. Optional interactions are specified by the stereotype `<<optionalInteraction>>` that extends the *Interaction* metaclass.
- **Variation.** A variation point in a PL sequence diagram means that for a given product, only one interaction variant defined by the variation point will be present in the derived sequence diagram. The *Interaction* encloses a set of sub-interactions, the variation mechanism can be specified by two stereotypes: `<<variation>>` and `<<variant>>`; the both stereotypes extend the *Interaction* metaclass. To

distinguish different variants in the same sequence diagram, we associate to the interaction stereotyped with <<variant>> a tagged value: {variation = Variation} to indicates its enclosing variation point(the enclosing interaction stereotyped with <<variation>>).

– **Virtuality.** A virtual part in a sequence diagram means that this part can be redefined for each product by another sequence diagram. The virtual part is defined using a stereotype <<virtual>> that extends the *Interaction* metaclass.

An interaction can be a variation point and a variant for another variation point in the same time. This means that it is enclosed in the interaction stereotyped <<variation>> and in the same time it encloses a set of interaction variants. In this situation, the interaction is defined with the two stereotypes: <<variation>> and <<variant>>.

Constraints. Structural constraints can be associated to the stereotypes and the tagged value defined above. For example, the constraint that concerns the <<variant>> stereotype and that specifies that each interaction stereotyped <<variant>> should be enclosed in one and only one interaction stereotyped <<variation>> can be formalized using OCL as an invariant to the <<variant>> stereotype:

context<<variant>>
inv: self.enclosingInteraction → select(oclIsKindOf(Variation)→ size()=1

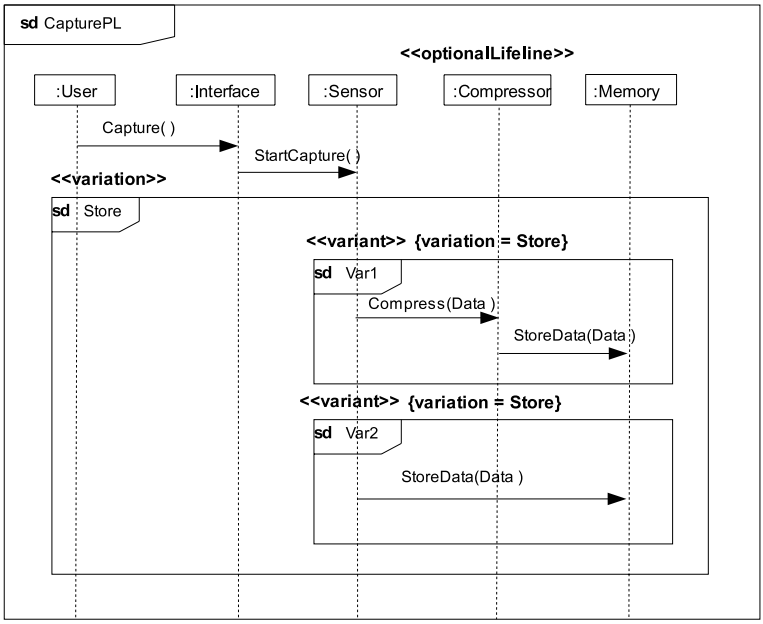

Fig. 3. The Sequence Diagram Capture

Example. Figure 2 shows the `CapturePL` sequence diagram that concerns the camera PL example. It illustrates the interaction to capture and to store data into the memory. This sequence diagram includes two types of variability: the presence of the `Compressor` object and the variation in the interaction `Store`. The `Compressor` lifeline is defined as optional, and the interaction `Store` (stereotyped `<<variation>>`) defines two variants interaction `Var1` and `Var2` (stereotyped `<<variant>>`) to store data into the memory. The first stores data after its compression and the second one stores them without compression. The tagged value {`variation = Store`} is added to the two interactions variants.

Table 1. Stereotypes and tagged values in the UML profile for PL

Stereotype/Tagged values	Applies to	Description
<<optional>>	Classifier, Package, Feature	Indicates that the element (classifier, package, or feature) is optional.
<< variation>>	Classifier	Indicates that the abstract class represents a variation point with a set of variants.
<<variant>>	Classifier	Indicates that a class is a variant of a variation point.
<<optionalLifeline>>	Lifeline	Indicates that the lifeline in the sequence diagram is optional.
<<optionalInteraction>>	Interaction	Indicates that the behavior described by the interaction is optional.
<<variation>>	Interaction	Indicates that the interaction is a variation point with two or more interaction variants.
<<variant>>	Interaction	Indicates that the interaction is a variant behavior in the context of a variation interaction.
<<virtual>>	Interaction	Indicates that the interaction is a virtual part.
{variation = Variation}	<<variant>>	Indicates the variation point related to this variant.

2.3 The Profile Structure

Table 1 summarizes the defined stereotypes and tagged values in the UML profile for PL. Figure 4 illustrates the structure of the proposed profile for PL (we follow notations for profiles as defined in [5]). Stereotypes are defined as class stereotyped with `<<stereotype>>`. UML metaclasses are defined as classes stereotyped with `<<metaclass>>`. Tagged values are defined as attributes for the defined stereotypes. The extensions proposed for class diagrams are defined in the `staticAspect` package, and thus for sequence diagrams are gathered in the `dynamicAspect` package. The `<<variant>>` stereotype in the `staticAspect` (respectively in the package

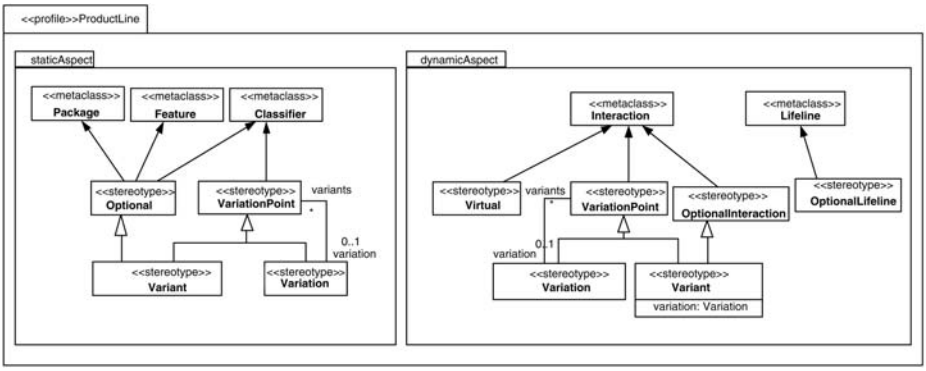

Fig. 4. UML profile for PL

`dynamicAspect`) inherits from the <<optional>> stereotype (respectively from the `<<optionalInteraction>>` stereotype). This means that each variant is optional too.

3 From PL Models to Product Models

A UML profile includes not only stereotypes, tagged values and constraints but also a set of operational rules that define how the profile can be used. These rules concern for example code generation from models that conform to this profile or model transformations. For the PL profile, this part can be used to define the product derivations as model transformations. A product derivation consists in generating from PL models the UML models of each product. The product line architecture is defined as a standard architecture with a set of constraints [2]. PL constraints guide the derivation process. In what follows we present two types of PL constraints: the generic constraints that apply to all PL, and specific constraints that concern a specific PL. We show how these constraints should be considered for the derivation process.

3.1 Generic Constraints

The introduction of variability, and more especially optionality in the UML class diagrams (specified by the `<<optional>>` stereotype), improves genericity but can generate some incoherences. For example, if a non-optional element depends on an optional one, the derivation can produce an incomplete product model. So the derivation process should *preserve* the coherence of the derived products. [12] proposes to formalize coherence constraints using OCL. Constraints that concern any PL model are called *Generic Constraints*. An example of such constraint is the dependency constraint that forces non optional elements to depend only on non optional elements. A dependency in the UML specifies a require relationship between two or more elements. It is represented in the UML meta-model [5] by the meta-class *Dependency*; it represents the relationship between a set of suppliers and clients. An example of the UML Dependency is the

"Usage", which appears when a package uses another one. The dependency constraints is specified using OCL as an invariant for the *Dependency* metaclass[1]:

context Dependency
inv:
self.supplier exists(S:ModelElement | S.isStereotyped ('optional')) **implies**
self.client **forAll**(C:ModelElement | C.isStereotyped('optional'))

While the <<variant>> stereotype inherits from the <<optional>> one (see Figure 4), the dependency constraint also is applied to variants. In the sens that a non-optional element can not depends on the variant one. The generic constraints may be seen as well-formedness rules for the UML modeled product line.

3.2 Specific Constraints

A fundamental characteristic of product lines is that all elements are not compatible. That is, the selection of one element may disable (or enable) the selection of others. For example in the sequence diagram `CapturePL` in Figure 4 the choice of the variant `Var1` in the specific product needs the presence of the compressor object. Dependencies between PL model elements are called *Specific Constraints*. They are associated to a specific product line and will be evaluated on all products derived from this PL. So another challenge for the product derivation is to *ensure* specific constraints in the derived products. These constraints can be formalized as OCL meta-level constrains [12]. The following constraint specifies the presence dependency in the sequence diagram `CapturePL` between the interaction variant `Var1`, and the `Compressor` lifeline. i.e: the presence of the interaction variant `Var1` requires the presence of the `Compressor` lifeline. It is added as an invariant to the *Interaction* metaclass:

context Interaction
inv: self.fragments → **exists** (I: IntercationFragment | I.name ='Var1') **implies**
self.lifeline → **exists** (L:Lifeline | L.name='Compressor')

In addition to the presence constraint, specific constraints include the mutual exclusion constraint. It expresses in a specific PL model that two optional classes cannot be present in the same product. This can be formalized using OCL, for example the mutual exclusion constraint between two optional classes called `C1` and `C2` in a specific PL is expressed using OCL as an invariant to the Model meta-class[2]:

context Model
inv:
self.presenceClass('C1') **implies not** self.presenceClass('C2'))
and (self.presenceClass('C2') **implies not** self.presenceClass('C1'))

[1] `isStereotyped(S) : boolean` is an auxiliary OCL operation indicates if an element is stereotyped by a string S.

[2] `presenceClass(C) : boolean` is an auxiliary OCL operation indicates if a class named C is present in a specific UML Model.

3.3 Product Models Derivation

The products derivation consists in generating from the PL models (UML class diagrams and sequence diagrams) models for each product. Product line models should satisfy generic constraints *before* the derivation while the derived product model should satisfy specific constraints. This means that generic constraints represent the pre-conditions of the derivation process and specific constraints represent the post - conditions for the derivation process:

DeriveProduct(PLModel : **Model**):**Model**
pre: –check generic constrains on PLModel
post: – check specific constraints on the derived product model.

Figure 5 shows the derived class diagram for a camera product. This product does not support the compression feature and only supports Interface 1 and Interface 2. It is obtained from the PL class diagram by removing the class `Compressor` and the class `Interface 3`. Figure 6 shows the derived `Capture` sequence diagram for this camera product. It is obtained by removing the `Compressor` lifeline and the choice of the `Var2` interaction (the `Var1` interaction is removed).

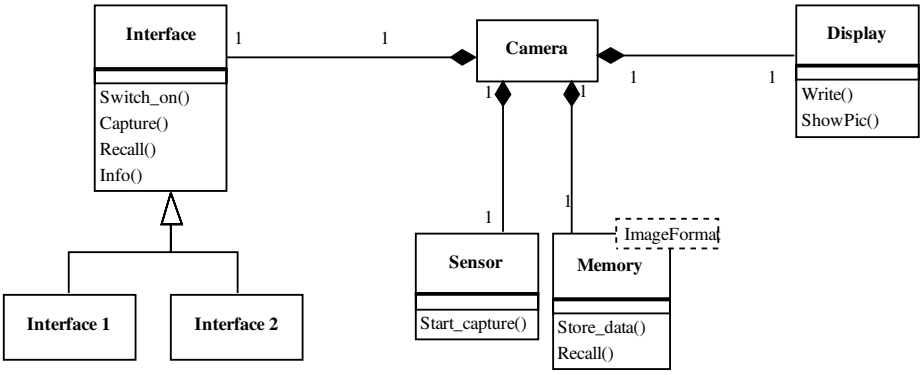

Fig. 5. The derived class diagram for a specific camera product

4 Related Work

Many work have studied modeling of PL variability using UML. [4] uses UML extensions mechanisms to specify variability in UML diagrams. However, despite the <<optional>> stereotype for UML statecharts and sequence diagrams, these extensions mainly focuses on the static aspects of the PL architecture. To model dynamic aspects of PLs, we have proposed three constructs to specify variability in sequence diagrams.

KobrA [1] is a method that combines product line engineering and component-based software development. It uses the UML to specify component. Variability is introduced

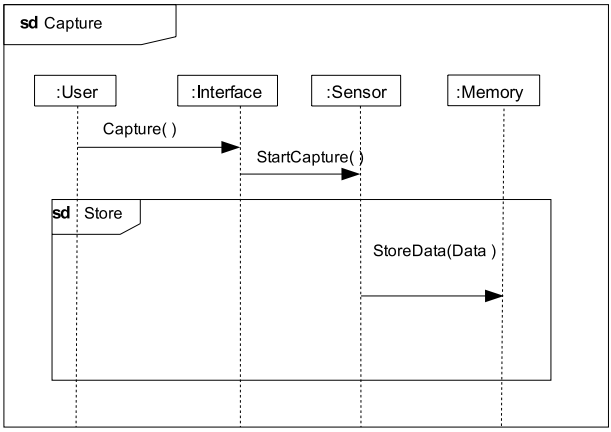

Fig. 6. The derived Sequence Diagram for a specific camera product

in the KobrA components using the <<variant>> stereotype. This stereotype is used to model any feature that are not common to all product. [3] proposes a set of UML extensions to describe product line variability. They only concern UML class diagrams. While we use OCL to model specific constraints, [3] models them using two stereotypes: <<require>> and <<mutex>> respectively for the presence and the exclusion mutuel constraint.

[9] proposes notations for product lines. They are gathered in the profile called UML-F. In fact this profile is defined for frameworks and it only concerns static aspects of the product line. [8] proposes a metamodel based on UML for product line architectures. Variability is introduced only in terms of alternatives.

5 Conclusion

In this paper, we have proposed a set of extensions as a UML profile for PL. These extensions concern UML class diagrams, and sequence diagrams. They are defined on the UML 2.0 metamodel. This profile is not yet implemented. We have only proposed some constraints, the definition of the defined profile should be refined with more constraints.

We intend to implement this profile with the UMLAUT. UMLAUT [6] is a framework for building tools dedicated to the manipulation of models described using the UML. A new version of the UMLAUT framework is currently under construction in the Triskell[3] team based on the MTL (Model Transformation Language), which is an extension of OCL with the MOF (Meta-Object Facility) architecture and side effect features, so it permits us to describe the process at the meta-level and to check OCL constraints. The MTL language can be used to define the derivation process.

[3] www.irisa.fr/triskell

References

1. Colin Atkinson, Joachim Bayer, Christian Bunse, Erik Kamsties, Oliver Laitenberger, Roland Laqua, Dirk Muthig, Barbara Paech, Jürgen Wüst, and Jörg Zettel. *Component-based Product Line Engineering with UML*. Component Software Series. Addison-Wesley, 2001.
2. Clements.P Bass.L and Kazman.R. *Software Architecture in Practices*. Addison-Wesley, 1998.
3. Matthias Clauß. Modeling variability with uml. In *GCSE 2001 Young Researchers Workshop*, 2001.
4. J.C Duenas, W. El Kaim, and Gacek C. Style, structure and views for handling commonalities and varibilities - esaps deliverable (wg 2.2.3). Technical report, ESAPS Project, 2001.
5. Object Management Group. Uinified modeling language specification version 2.0: Superstructure. Technical Report pct/03-08-02, OMG, 2003.
6. Wai-Ming Ho, Jean-Marc Jézéquel, Alain Le Guennec, and François Pennaneac'h. UMLAUT: an extendible UML transformation framework. In *Proc. Automated Software Engineering, ASE'99, Florida*, October 1999.
7. ITU-T. Z.120 : Message sequence charts (MSC), november 1999.
8. Dirk Muthig and Colin Atkinson. Model-driven product line architectures. In Gary J. Chastek, editor, *Software Product Lines, Second International Conference, SPLC 2, San Diego, CA, USA, August 19-22, 2002, Proceedings*, volume 2379 of *Lecture Notes in Computer Science*. Springer, 2002.
9. Wolfgang Pree, Marcus Fontoura, and Bernhard Rumpe. Product line annotations with uml-f. In Gary J. Chastek, editor, *Software Product Lines, Second International Conference, SPLC 2, San Diego, CA, USA, August 19-22, 2002, Proceedings*, volume 2379 of *Lecture Notes in Computer Science*. Springer, 2002.
10. J. Warmer and A. Kleppe. *The Object Constraint Language-Precise Modeling with UML*. Object Technology Series. Addison-Wesley, 1998.
11. Tewfik Ziadi, Loïc Hélouët, and Jean-Marc Jézéquel. Modeling behaviors in Product Lines. In *Proceedings of Requirement Engineering for Product Lines Workshop (REPL02)*, pages 33–38, September 2002.
12. Tewfik Ziadi, Jean-Marc Jézéquel, and Frédéric Fondement. Product line derivation with uml. In Jilles van Gurp and Jan Bosh, editors, *Proceedings Software Variability Management Workshop*, pages 94–102. University of Groningen Departement of Mathematics and Computing Science, 2003.