Multi-Variant Modeling*

Concepts, Issues, and Challenges

Bernhard Westfechtel[1] and Reidar Conradi[2]

[1] Angewandte Informatik 1, Universität Bayreuth
D-95440 Bayreuth
Bernhard.Westfechtel@uni-bayreuth.de
[2] Norwegian University of Science and Technology (NTNU)
N-7491 Trondheim, Norway
Reidar.Conradi@idi.ntnu.no

Abstract. When applying model-driven engineering to a product line, there is a need to deal with multi-variant models. So far, in industry software product line engineering has primarily been applied to data represented in (text) files and directories. Applying variation to non-textual models is more difficult, since models are complex structured objects. This paper presents basic concepts and discusses some issues and challenges with respect to multi-variant modeling.

1 Introduction

Model-driven engineering (MDE) denotes an approach to software development which strongly emphasizes the use of explicit and formal models throughout the whole software lifecycle. Models do not merely serve as documentation. Rather, MDE aims at developing executable models, eliminating the need for manual programming.

According to [1], *software product line engineering* (SPLE) is a paradigm to develop software applications using platforms and mass customization. In this context, the variability model plays a central role because it defines the dimensions of variation supported by the product line.

So far, SPLE has been applied primarily to data represented as (text) files and directories. This picture is beginning to change. *Model-driven product line engineering* (MPLE) has been addressed in several research projects [2–4], and initial tool support has been provided by industry [5].

Nevertheless, the integration of model-driven and product line engineering still has to be explored further; the field has not yet reached maturity. In this paper, we focus on *multi-variant modeling*, i.e., the construction and representation of models incorporating multiple variants. We present concepts, issues, and challenges of multi-variant modeling, and investigate the state of tool support.

* in: Mira Mezini, Danilo Beuche, Ana Moreira (Eds.): Proceedings 1st International Workshop on Model-Driven Product Line Engineering (MDPLE 2009), CTIT Proceedings, Twente, The Netherlands, 57–67 (June 2009), http://www.feasiple.de/workshop_en.html

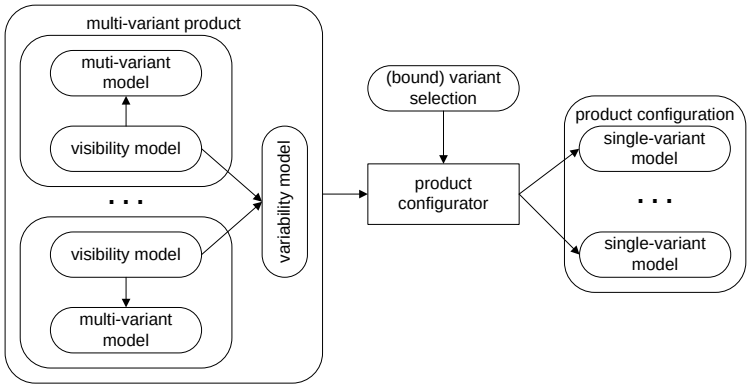

Fig. 1. Framework for multi-variant modeling

2 Multi-Variant Modeling

2.1 Conceptual Framework

Figure 1 introduces a simple *conceptual framework* for multi-variant modeling and product configuration. A *multi-variant software product* is the *union of all variants* of the software product. It is composed of a set of interrelated *multi-variant models*. A global *variability model* defines the variation points and variants the product line is required to support. For each multi-variant model, a *visibility model* defines in which variants the elements of the model are contained. To obtain a specific product variant, a bound *variant selection* is defined. Thus, for each variation point a specific variant is selected. The *product configurator* evaluates the visibilities of the elements of multi-variant models against the variant selection. It selects only those elements which are visible under the current selection. Thus, the product configurator produces a set of *single-variant models*, which are subsets of the underlying multi-variant models.

2.2 Variability Model

A *variability model* describes the variation points along which a software product may vary, as well as supported combinations of variants (e.g., Feature-Oriented Domain Analysis [6] or the Orthogonal Variability Model [1]). In the context of this paper, we define a variability model as a relation $vm \subseteq vp_1 \times \ldots \times vp_m$, where each vp_i represents a variation point[3]. The relation vm could be defined *extensionally*, i.e., by enumerating all tuples in vm. Usually, vm is defined *intensionally* by some predicate vp such that $vm = \{(v_1, \ldots v_m) | vp(v_1, \ldots v_m)\}$. A *variant selection* is a subset $vs \subseteq vm$. The selection is *bound* if $|vs| = 1$; a bound selection corresponds to a single *product variant*.

[3] For feature models, it would be more appropriate to define a variant as a set rather than a tuple of features and the variability model as a set of variants. However, this difference is immaterial to the discussion below.

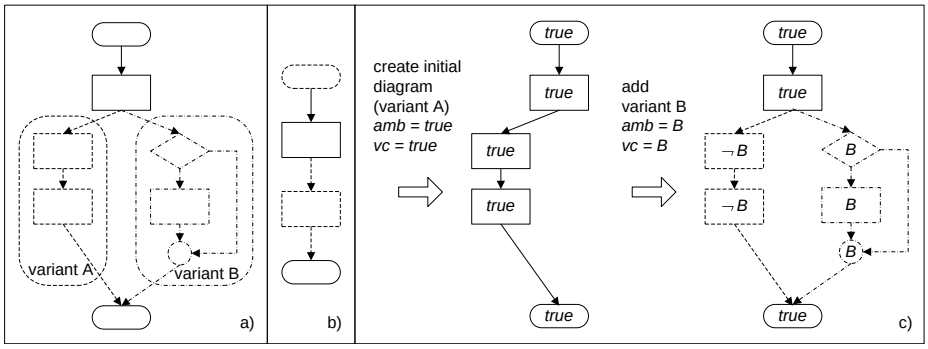

Fig. 2. Example: flow diagrams

2.3 Single- and Multi-Variant Models

In the context of MDE, a *model* is an abstraction of the system to be built. A model m is represented as a pair (E, S), where E denotes a set of a elements and S is a structure defined on these elements. A model is an instance of some *metamodel*, which defines the types of available model elements (e.g., data objects, composition relations) and the constraints imposed on their combinations (e.g., by class diagrams and OCL constraints). We may view a metamodel mm as a set of (potential) models conforming to this metamodel, i.e., $mm = \{m | mp(m)\}$, where mp denotes a (complex) predicate to be satisfied.

The simplest way to represent a *multi-variant model* is to reuse the metamodel of *single-variant models*, i.e., single- and multi-variant models are instances of the same metamodel mm. This approach assumes that the metamodel mm has been defined in such a way that variation may be expressed under the constraints of mm. Consider, e.g., class diagrams, where variation may be expressed by disjoint subclassing or by realizing interfaces by alternative classes. This kind of variability is called *internal*. In fact, many approaches to SPLE tacitly assume that it suffices to exploit internal variability [4, 2, 7, 1]. Unfortunately, not all metamodels support internal variability:

Example 1 (Flow diagrams). A (sequential) *flow diagram* is a connected graph which consists of a single start node (no incoming, one outgoing control flow, cardinality 0/1), as well as multiple activity nodes (1/1), binary decision nodes (1/2), join nodes (+/1), and end nodes (1/0). A multi-variant control flow diagram is given in Figure 2a. Here, solid and dashed lines denote universal (mandatory) and variant-specific (optional) parts, respectively. The multi-variant control flow diagram is not a valid control flow diagram: The first activity node has more than one outgoing edge, and the end node has more than one incoming edge. □

A multi-variant model is a *union of single-variant models*. In general, we cannot expect that such a union is a valid instance of a single-variant metamodel. In particular, cardinality constraints for single-variant models may not hold for their superimposition.

In *universal multi-variant modeling*, all model elements should be allowed to vary (regardless of whether the modeler will apply variation universally or only with certain methodological constraints). This can be achieved only by *external variability*, i.e., by a "meta-level" mechanism universally applied to any kind of model. Such mechanisms are provided e.g. by *software configuration management systems*, which add version control to software objects.

2.4 Visibility Models

Each multi-variant model is associated with a *visibility model* which defines the visibilities of the model elements. We may consider a multi-variant model as a set of pairs $(e, vis(e))$, where e denotes an element and $vis(e)$ its visibility. Visibilities may be represented in different ways. For example, in the UML stereotypes, tagged values, or annotations may be used to express visibilities. Formally, the visibility model may be defined as a function vis which maps each model element onto the set of variants in which it is visible: $vis : E \rightarrow 2^{vm}$, where $vis(e) = vs$ is some (not necessarily bound) variant selection. The visibility of a product element e may be presented by some *visibility predicate* $visp(e)$ such that $vis(e) = \{(v_1, \ldots, v_m) | visp(e)(v_1, \ldots, v_m)\}$.

Let e_1 and e_2 denote two variants of the "same" element (which assumes some sameness criterion, e.g., element identifiers). The visibilities of different elements must be disjoint:

$$e_1 \neq e_2 \Rightarrow vis(e_1) \cap vis(e_2) = \emptyset \tag{1}$$

2.5 Product Configuration

The *product configurator* takes a set of multi-variant models decorated with visibilities and a variant selection. The variant selection should be consistent, i.e., $vs \subseteq vm$ or (in terms of predicates vsp for the variant selection and vp for the variability model, as defined in Subsection 2.2):

$$vsp \Rightarrow vp \tag{2}$$

The product configurator returns a set of models whose elements are *visible* under the current selection. If the variant selection is bound, for each element at most one variant will be selected. In this case (which we will assume in the following), the product configurator returns a set of *single-variant models*.

An element is selected if its visibility contains the variant selection. In terms of predicates, this may be rephrased as follows:

$$vsp \Rightarrow visp(e) \tag{3}$$

The configured models should be *valid* instances of the corresponding single-variant metamodels. That is, the product configurator should produce a *syntactically consistent result* (which is a necessary, yet not sufficient condition that the result is usable as it stands). Unfortunately, this requirement is hard to guarantee in general. Counterexamples may be constructed easily:

Example 2 (Inconsistencies in configured models). Three errors are contained in the multi-variant flow diagram of Figure 2b:

1. The mandatory start node has been marked as optional (variant specific).
2. Its outgoing control flow has been marked as mandatory, even though the start node is optional (this may give rise to a dangling control flow).
3. If the second activity node is omitted (with its adjacent control flows), the diagram is no longer connected. □

Thus, *constraints* must be imposed on the combination of visibilities, e.g., to preserve product properties (well-formedness rules). For example, the first error in Example 2 may be avoided by requiring that a mandatory element must not be marked as optional. More precisely, the visibility of a mandatory component e_1 (the start node) must be implied by the visibility of the enclosing composite e_2 (the flow diagram):

$$visp(e_2) \Rightarrow visp(e_1) \tag{4}$$

The second error may be avoided by a constraint referring to the *dependencies* between model elements. For some model $m = (E, S)$, the structure S on its elements E induces a dependency relation $D \subseteq E \times E$, where $d(e_1, e_2)$ holds if and only if e_1 depends on e_2. The visibility of a dependent element e_1 must not exceed the visibility of its master e_2:

$$visp(e_1) \Rightarrow visp(e_2) \tag{5}$$

Since a mandatory component existentially depends on its enclosing composite, the equations above jointly imply that their visibilities must be equal.

The *product constraints* defined above are necessary, yet not sufficient conditions for product consistency. In general, arbitrarily complex constraints may be defined in the metamodel. This is exemplified by the third error (the diagram is not connected).

2.6 Multi-Variant Editing

In the previous subsection, we have discussed product configuration, tacitly assuming that a set of multi-variant models has already been created. In the following, we will address the question how multi-variant models come into being. This requires some way of *multi-variant editing*.

In the case of *unfiltered editing*, the user edits a multi-variant model, where all variants are displayed and modified simultaneously. This corresponds to editing a program file with conditional compilation statements. The advantage of unfiltered editing consists in the fact that the user sees *all variants simultaneously* and therefore may assess the impact of changes better than in the case of filtered editing to be explained below. On the other hand, the information overload incurred by the overlay of multiple variants may be difficult to manage.

In the case of *filtered editing*, the user edits a single variant called *view*. Usually, it is desired that editing may affect more than just the variant displayed in the view. One way to deal with this problem is to define an *edit set* which defines the scope of a change [8]. A view is a bound variant selection vs, and the edit set is a set of variants es containing vs. All elements are affected whose visibilities overlap with the edit set.

An alternative approach of filtered editing makes use of *layers* or *change sets* [9]. The user defines a single-variant view by a sequence of layers. Editing operations refer to some designated layer, into which the performed changes are aggregated. The scope of the changes is confined to the designated layer, which can be used to construct multiple variants.

3 The UVM Approach

3.1 Background

This section briefly presents the UVM approach as an example for multi-variant management. UVM, which stands for *Uniform Version Model* [10], provides for extrinsic variability. UVM offers a basic way to express variant handling, independent of (orthogonal to) the data or product model and the application of such models in a user context. The approach has been applied to both textual and non-textual data (e.g., entity-relationship diagrams [11]).

UVM was developed in the context of *software configuration management* (SCM). The SCM landscape is dominated by systems which focus on temporal and cooperative versioning (usually based on version graphs, see e.g. Subversion [12], CVS [13] or ClearCase [14]) and provide only limited support for logical versioning (variants). For this reason, SCM and SPLE are often perceived as more or less disjoint disciplines. For example, BigLever Software Gears covers only product line variants and assumes that revisions of the product line are managed by an SCM system [15].

However, work on multi-dimensional variation has been performed in several SCM research prototypes, as well [16, 17]. Moreover, a few research projects in SCM were dedicated to the development of a *uniform version model*, supporting all dimensions of evolution through a single base mechanism (ICE [18] and UVM [10]).

3.2 Basic Versioning

In UVM, each stored and versioned information *item* (with application-specific granularity and interpretation) has a *visibility (predicate)* tag called $visp$. There are as many pairs of (visibility-value,item-value) as there are versions of this item. The visibility is a logical expression in 3-nary logic over an open set of global versioning attributes.

UVM is based on *filtered editing*. The view is defined by a *version choice*, i.e. a bound combination of versioning attributes. Depending on some version choice vc, the visibilities will evaluate to $false$ or $true$. This assumes, however, that all attributes are completely bound, i.e., no unbound version settings left. The single-variant view is then the (sequential) union of item-values with visibilities $visp = true$, and application-specific tools will only see this visible subset.

The edit set is defined by an *ambition*, i.e., a logical expression with a partial binding of versioning attributes. The version choice must lie inside the ambition:

$$vc \Rightarrow amb \tag{6}$$

The edit set defines the scope of the change. The visibilities of elements are managed automatically. All new elements e inherit their visibilities from the ambition:

$$visp_{new}(e) = amb \tag{7}$$

Deletion of an element does not mean that the element is physically removed. Rather, all versions of the element are no longer visible under the edit set (ambition). This is achieved by constraining the visibilities:

$$visp_{new}(e) = visp_{old}(e) \wedge \neg amb \tag{8}$$

Example 3 (Multi-variant editing).

An example is given in Figure 2c. In the first step, the predicates for the edit set and the view are both set to $true$. This results in an initial variant of the flow diagram, where all elements carry the visibility $true$. For the nodes, the visibility is shown inside the node. Edges carry visibilities, as well, but their visibilities are not displayed. The flow diagram created so far corresponds to variant A in Figure 2a. The second step is performed under the visibility B both for the view and the edit set. All elements created so far qualify for the view. In the view, two activities (on the left) are deleted, and a new branch of the control flow diagram is inserted (on the right). The new elements get visibility B, the visibility of the deleted elements is constrained to $\neg B$. After the second step, we obtain the multi-variant flow diagram of Figure 2a. Variant A is represented implicitly (as $\neg B$). □

3.3 Consistency Control

UVM supports consistency control through the following mechanisms:

Automated management of visibilities All performed changes are tagged consistently with visibilities according to the rules given above. Thus, many errors can be avoided which the user may introduce by defining visibilities individually and manually.

Enforcement of product constraints In a data model layer above the core, product constraints may be implemented. For example, the visibility of an association may be narrowed down automatically (i.e., it is not visible if one of its ends is not visible) [11].

Enforcement of version constraints The user may define constraints on the combination of versioning attributes (e.g., the variants $Windows$ and $Linux$ are mutually exclusive). These constraints are exploited by a versioning assistant which supports the user in consistent version selections.

However, UVM cannot guarantee in general that a new version choice gives us a single-variant that is consistent, either syntactically or wrt. static or dynamic semantics. So a *merge-edit operation* may be needed when two attribute settings are combined in a new version choice. In this way, a *feedback cycle* is realized: By editing a configured variant, the capabilities of the product line are enhanced.

Example 4 (Merge-edit to reconcile mutually conflicting items from overlapping versions).

Consider a class diagram containing some class C, where two users have both added some method called m in parallel. This results in two method variants m_1 and m_2 with visibilities $visp_1$ and $visp_2$, respectively. Now, both variants are combined (automatic *merge step*). This is achieved by the version choice $visp_1 \wedge visp_2$, which is also the ambition. Since the version choice implies both visibilities (e.g., $visp_1 \wedge visp_2 \Rightarrow visp_1$), both method variants m_1 and m_2 are selected simultaneously. This results in a name clash: Method m is declared twice.

The user may now resolve the conflict in any desired way (e.g., by deleting or renaming one of the method variants, or by merging both variants manually into a single combined variant). This *manual edit step* will eventually result in a consistent class diagram (and associated implementation). When the changes are committed, the visibilities are updated for the changed elements. For example, in the case of a manual merge, a new method variant m_3 is created with visibility $visp_1 \wedge visp_2$, and the visibilities of the old variants are narrowed down. For example, the new visibility of m_1 will be $visp_1 \wedge \neg(visp_1 \wedge visp_2) = visp_1 \wedge \neg visp_2$. Thus, the old variants will not be selected any more under the combination $visp_1 \wedge visp_2$. □

---

4 Related Work

Table 1 compares UVM against a few tools for multi-variant modeling.

Feature Mapper [19, 4] is based on feature modeling and supports the annotation of EMF models with visibilities. The user may switch between single- and multi-variant views. Visibilities may be defined individually for each model element. Feature Mapper also offers a recording mode where a visibility is defined beforehand and all elements created are decorated automatically with this visibility. Consistency control is not addressed; Feature Mapper cannot guarantee (or check) that a configured model variant is syntactically consistent.

Feature-based model templates were introduced in [7]. Cardinality-based feature models [20] are used for the variability model. The approach refers to models whose metamodels are defined with MOF and OCL. Product consistency of configured variants may be checked automatically [21]: By an abstract interpretation of well-formedness rules given in OCL for the respective metamodel, it can be checked whether each variant which may be configured according to a given feature model is consistent with respect to these rules. In this respect, feature-based model templates go beyond the capabilities of competing approaches.

	Feature Mapper	**Feature-based model templates**	**EASEL**	**UVM**
Domain	EMF models	MOF- and OCL-defined models	Class diagrams	(generic)
Variability	internal	internal	external	external
Variability model	feature model	feature model	layers	three-valued logic
Variability constraints	not supported	excludes and requires	implications	logical expression
Multi-variant representation	interleaved deltas	interleaved deltas	directed deltas	interleaved deltas
Visibilities	logical expression	logical expression	defined by layers	logical expression
Visibility management	manual or automatic	manual	automatic	automatic
Multi-variant editing	filtered or unfiltered	unfiltered	filtered	filtered
Product consistency	not addressed	verifiable	checked, inconsistencies tolerated	(in data model layer)

Table 1. Classification of tools for multi-variant modeling

EASEL [22] supports multi-variant editing for class diagrams. In contrast to the other approaches compared here, EASEL relies on layers, i.e., directed deltas rather than interleaved deltas. Instead of decorating model elements with visibilities, changes (addition, deletion, and modification of model elements) are aggregated into layers which may be composed dynamically. EASEL detects contradictory or inconsistent combinations of layers automatically by analyzing the change operations contained in the layers. Inconsistencies are tolerated, and may be fixed by the user in a similar way as in UVM (merge-edit).

5 Conclusion

In this paper, we have examined multi-variant modeling, and we have briefly discussed how an approach from SCM (the Uniform Version Model) may be applied to multi-variant modeling. From our examination, we draw the following conclusions:

– Multi-variant modeling should be supported in a universal and uniform way. *Universal* means that each item of a model (elements, attributes, references, etc.) should be allowed to vary. *Uniform* means that multi-variant modeling should be supported by one generic base mechanism which may be applied to any kind of model, independently of the underlying metamodel. In a layered architecture, we may push this argument even further and strive to provide a base layer which is even independent of the metametamodel (and corresponding data model) [10].

- A multi-variant model is not necessarily just an ordinary model. Annotating ordinary models with visibilities is limited to exploiting *internal variability*. For universal multi-variant modeling, *external variability* is required.
- In the case of external variability, the *union of single-variant models* results in a multi-variant model for which *single-variant constraints* might be *violated*. That is, a multi-variant model is not necessarily a valid instance of a single-variant metamodel.
- No matter whether internal or external variability is applied: In general, it is hard to guarantee even the *syntactic consistency* of *configured single-variant models*. Some errors may be eliminated by appropriate management of visibilities, but this is unlikely to work in all cases. As a consequence, modeling tools should be more "forgiving", i.e., they should be capable of processing inconsistent models; otherwise, the user cannot conveniently check and fix the result produced by the configurator.
- *Simple and intuitive tools* are required for assisting the user in managing the complexities of multi-variant modeling. Unfiltered editing raises the risk of information overload. However, filtered editing exhibits some pitfalls, as well (e.g., adequate definition of the scope of a change). More research is required on adequate user interfaces.
- Even the best tools will not eliminate the complexity of multi-variant modeling. Thus, a *process* is required which ensures disciplined use of the concepts.

Acknowledgments The authors gratefully acknowledge the constructive comments of the unknown reviewers.

References

1. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer, Berlin, Germany (2005)
2. Bayer, J., Gerard, S., Haugen, Ø., Mansell, J., Møller-Pedersen, B., Oldevik, J., Tessier, P., Thibault, J.P., Widen, T.: Consolidated product line variability modeling. In Käköla, T., Dueñas, J.C., eds.: Software Product Lines: Research Issues in Engineering and Management. Springer, Berlin (2006) 195–241
3. Stephan, M., Antkiewicz, M.: Ecore.fmp: A tool for editing and instantiating class models as feature models. Technical Report 2008-08, University of Waterloo, Waterloo, Canada (2008)
4. Heidenreich, F., Kopcsek, J., Wende, C.: Featuremapper: Mapping features to models. In: Companion Proceedings of the 30th International Conference on Software Engineering (ICSE'08), Leipzig, Germany, ACM Press (May 2008) 943–944
5. Krueger, C.W.: Leveraging the synergy of model-driven development and software product line engineering. Technical Report #200710311, BigLever Software, Austin, TX (October 2007)
6. Chang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania (November 1990)
7. Czarnecki, K., Antkiewicz, M.: Mapping features to models: A template approach based on superimposed variants. In Glück, R., Lowry, M.R., eds.: 4th International Conference on Generative Programming and Component Engineering (GPCE 2005). LNCS 3676, Tallinn, Estonia, Springer (October 2005) 422–437

8. Sarnak, N., Bernstein, R., Kruskal, V.: Creation and maintenance of multiple versions. In Winkler, J.F.H., ed.: Proceedings of the International Workshop on Software Version and Configuration Control, Grassau, Germany, Teubner Verlag (1988) 264–275
9. Goldstein, I.P., Bobrow, D.G.: A layered approach to software design. Technical Report CSL-80-5, XEROX PARC, Palo Alto, California (1980)
10. Westfechtel, B., Munch, B.P., Conradi, R.: A layered architecture for uniform version management. IEEE Transactions on Software Engineering **27**(12) (December 2001) 1111–1133
11. Munch, B.P.: Versioning in a software engineering database — the change oriented way. PhD thesis, NTNU Trondheim, Norway (1993) IDT-Report 1993:4, 265 p.
12. Collins-Sussman, B., Fitzpatrick, B.W., Pilato, C.M.: Version Control with Subversion. O'Reilly & Associates, Sebastopol, California (2004)
13. Vesperman, J.: Essential CVS. O'Reilly & Associates, Sebastopol, California (2006)
14. White, B.A.: Software Configuration Management Strategies and Rational ClearCase. Object Technology Series. Addison-Wesley, Reading, Massachusetts (2003)
15. Krueger, C.W.: The software product line lifecycle framework. Technical Report #200805071r1, BigLever Software, Austin, TX (December 2008)
16. Mahler, A.: Variants: Keeping things together and telling them apart. In Tichy, W.F., ed.: Configuration Management. Volume 2 of Trends in Software. John Wiley & Sons, New York (1994) 73–98
17. Tryggeseth, E., Gulla, B., Conradi, R.: Modelling systems with variability using the PROTEUS configuration language. In Estublier, J., ed.: Software Configuration Management: Selected Papers SCM-4 and SCM-5. LNCS 1005, Seattle, WA, Springer (April 1995) 216–240
18. Zeller, A., Snelting, G.: Unified versioning through feature logic. ACM Transactions on Software Engineering and Methodology **6**(4) (October 1997) 397–440
19. Heidenreich, F., Wende, C.: Bridging the gap between features and models. In: Proceedings of the Second Workshop on Aspect-Oriented Product Line Engineering (AOPLE 07)
20. Czarnecki, K., Helsen, S., Eisenecker, U.W.: Formalizing cardinality-based feature models and their specialization. Software Process: Improvement and Practice **10**(1) (2005) 7–29
21. Czarnecki, K., Pietroszek, K.: Verifying feature-based model templates against well-formedness OCL constraints. In Jarzabek, S., Schmidt, D.C., Veldhuizen, T.L., eds.: Proceedings 5th International Conference on Generative Programming and Component Engineering (GPCE 2006), Portland, Oregon, ACM Press (October 2006) 211–220
22. Hendrickson, S.A., Jett, B., van der Hoek, A.: Layered class diagrams: Supporting the design process. In Nierstrasz, O., Whittle, J., Harel, D., Reggio, G., eds.: Proceedings 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006). LNCS 4199, Genova, Italy, Springer (October 2006) 722–736