FeaturePlugin: Feature Modeling Plug-In for Eclipse*

Michał Antkiewicz, Krzysztof Czarnecki
University of Waterloo
200 University Ave. West
Waterloo, ON N2V 1C3, Canada
{mantkiew,kczarnec}@swen.uwaterloo.ca

ABSTRACT

Feature modeling is a key technique used in product-line development to model commonalities and variabilities of product-line members. In this paper, we present *FeaturePlugin*, a feature modeling plug-in for Eclipse. The tool supports cardinality-based feature modeling, specialization of feature diagrams, and configuration based on feature diagrams.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications—*Tools*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Computer-aided software engineering (CASE)*

General Terms

Design

Keywords

Configuration, domain analysis, software-product lines, software reuse, system families, variability modeling and management

1. INTRODUCTION

Development of reusable software requires a product-line perspective, in which a product-line member is built from a common set of reusable assets [6, 24]. Feature modeling allows us to model the common and variable properties of product-line members throughout all stages of product-line engineering. At an early stage, feature modeling enables product-line scoping, i.e., deciding which features should be supported by a product line and which should not. In design, the points and ranges of variation captured in feature models need to be mapped to a common product-line architecture. Furthermore, feature models allow us to scope and derive domain-specific languages, which are used to specify product-line members in generative software development [7, 9]. Finally, feature models are also useful in product development as a basis for estimating development cost and effort, and automated or manual product derivation.

This paper describes *FeaturePlugin*, an Eclipse plug-in for feature modeling. While bringing Eclipse closer to software product-line and generative development communities, providing tool support for feature modeling as an Eclipse plug-in is particularly attractive for two reasons. First, integrating feature modeling as part of a development environment helps to optimally support modeling variability in different artifacts. This is important since variability permeates all artifacts and aspects of a product line, including implementation code, models, documentation, development process guidance, languages, libraries, and more. Secondly, we were able to generate the basic structure of FeaturePlugin using the generator for tree-oriented model editors provided by the Eclipse Modeling Framework (EMF), which significantly reduced our development effort. This was possible because feature modeling follows a tree structure.

The remainder of this paper is organized in six sections. Section 2 gives a brief introduction to feature modeling using an example. Support for configuration based on feature models is described in Section 3. Defining and checking additional constraints is covered in Section 4. Section 5 presents the feature-diagram specialization capability of the plug-in. Section 6 describes the support for user-defined annotations. Related work is given in Section 7. Section 8 concludes with a discussion and ideas for future work.

2. FEATURE MODELING

Feature modeling was proposed as part of the Feature-Oriented Domain Analysis (FODA) method [13], and since then, it has been applied in a number of domains including telecom systems [12, 16], template libraries [9], network protocols [1], and embedded systems [8]. Our plug-in implements *cardinality-based feature modeling* [10], which extends the original feature modeling from FODA with feature and group cardinalities, feature attributes, feature diagram references, and user-defined annotations.

A *feature* is a system property that is relevant to some stakeholder and is used to capture commonalities or discriminate among products in a product line. A *feature model* consists of one or more *feature diagrams*, which organize features into hierarchies. Figure 1 shows a sample feature model as it appears in the editing view of the plug-in. The model describes a product line of electronic shops. It consists of four feature diagrams, each having a *root feature*

*Supported by NSERC Discovery and IBM Eclipse Innovation Grants.

OOPSLA'04 Eclipse Technology eXchange (ETX) Workshop, Oct. 24-28, 2004, Vancouver, British Columbia, Canada.

indicated by the root symbol. The root feature `Payment` has two subfeatures: `PaymentTypes` and `FraudDetection`. The symbol indicates that `PaymentTypes` has a feature cardinality of [1..1]. Feature cardinality is an interval denoting how often a subfeature (and any possible subtree) can be cloned as a child of its parent when specifying a concrete system. The cardinality of [1..1] for `PaymentTypes` means that the payment aspect of a system must include payment types at least and at most once. In contrast, `FraudDetection` is optional, i.e., its cardinality is [0..1], as indicated by the symbol. The available payment types, i.e., `CreditCard`, `DebitCard`, and `PurchaseOrder`, are members of a *feature group*. The group symbol indicates group cardinality $\langle 1\text{–}k \rangle$, where k is the group size. Thus, our shop can support any non-empty subset of the three payment types. The other group symbol indicates group cardinality of $\langle 1\text{–}1 \rangle$, unless the group cardinality is stated explicitly. For example, `Expiration` has a group with cardinality $\langle 1\text{–}1 \rangle$, whereas `Chars` has a group with cardinality $\langle 2\text{–}4 \rangle$ (meaning that at least two different character types need to be used in a password). *Grouped features* are indicated by the symbol. Note that, in contrast to *solitary features* (i.e., subfeatures that are not grouped), grouped features do not have feature cardinalities. A feature can have an *attribute*, in which case the attribute type is indicated in parenthesis, e.g., `InDays(Integer)`. If a feature has cardinality other than [0..1] or [1..1], the cardinality is specified next to the feature, e.g., `Method`. Observe that the cardinality of `Method` has $*$ as upper bound, which means that `Method` can be cloned an unbounded number of times. Thus, a shop can have one or more custom shipment methods with individual rates. Finally, the root feature `EShop` has *references* to the three other diagrams, e.g., `ref:Payment` refers to the diagram with the root feature `Payment`. Note that references have cardinalities, too. For example, `ref:Shipping` is optional.

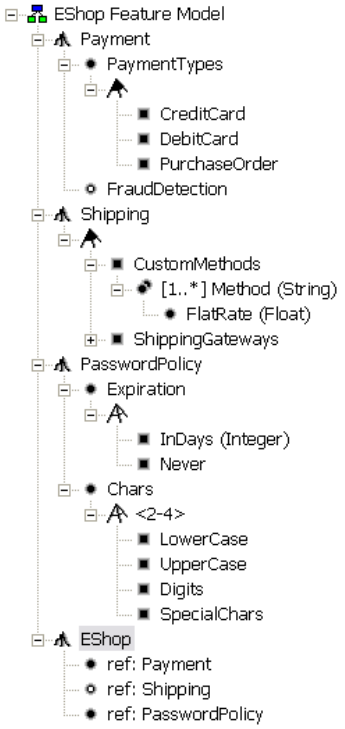

Figure 1: Example of a feature model in editor view

3. FEATURE-BASED CONFIGURATION

Configuration is the process of deriving a concrete configuration conforming to a feature diagram by selecting and cloning features, and specifying attribute values [10]. Figure 2 shows a configuration of the `EShop` feature diagram from Figure 1. Note that references to feature diagrams have been automatically unfolded. Check boxes appear next to optional features and grouped features to allow their selection. Features with cardinality whose upper bound is greater than 1 can be cloned. For example, `Method` was cloned two times, which automatically updated its cardinality from [1..*] to [0..*]. Also, attribute values for the two clones of `Method` and `FlatRate` as well as for `InDays` were specified.

Alternatively, a configuration can also be specified using a configuration wizard (see Figure 3). The wizard traverses the feature diagram top-down and left-to-right and offers the available configuration choices within the diagram on separate pages. A configuration specified using the wizard can also be viewed and fine-tuned using the tree-based interface as in Figure 2.

The resulting configuration can be used as input to code generators (as shown in [8]), or it can be accessed by a system as a runtime configuration. For this purpose, the configuration can be written out as an XML file, or it can be

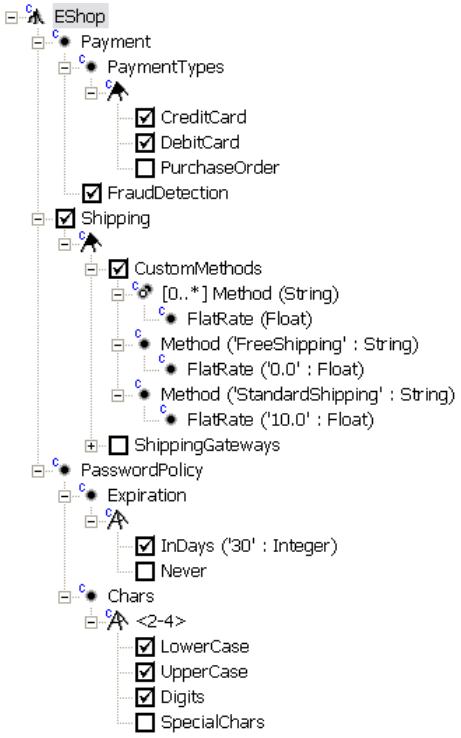

Figure 2: Configuration of `EShop` from Figure 1

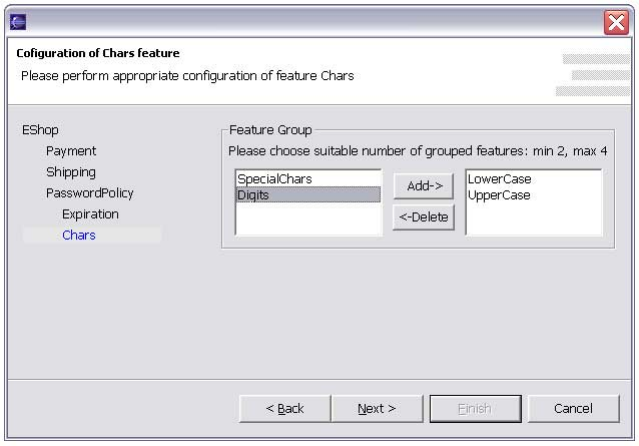

Figure 3: Wizard-based configuration of `EShop` from Figure 1

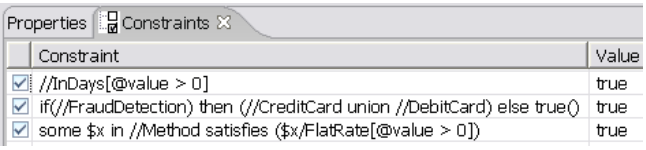

Figure 4: Constraints evaluated on the configuration from Figure 2

accessed in memory as an EMF model. When a configuration is exported to XML, features whose check boxes where not checked or whose cardinality is $[0..n]$ are ignored, as they are not considered to part of the configuration.

4. SPECIFYING AND CHECKING ADDITIONAL CONSTRAINTS

Feature models often need to contain *additional constraints*, i.e., those that cannot be expressed as feature or group cardinalities. Common examples are *implies* and *excludes* constraints. In general, additional constraints in cardinality-based feature models require tree-oriented navigation and query facilities, iteration mechanisms or quantifiers, and ways of counting feature clones in the scope of a given feature within a configuration. Furthermore, logic, arithmetic, set, and string operators on feature attributes and feature sets are desirable. Such constraints can be adequately expressed using XPath 2.0 [25]. If necessary, XPath can be easily extended with user-defined functions.

Figure 4 shows a few examples. The first constraint is an example of a local constraint requiring that the attribute of `InDays` is positive. The second constraint involves several features and states that selecting `FraudDetection` implies that `CreditCard` and/or `DebitCard` are selected (assuming that fraud detection is applicable to electronic payment only). The third constraint existentially quantifies over feature clones and states that at least one custom shipping method should have a rate of more than 0.

FeaturePlugin can check the additional constraints for a given configuration. For example, the configuration in Figure 2 satisfies all the constraints from Figure 4.

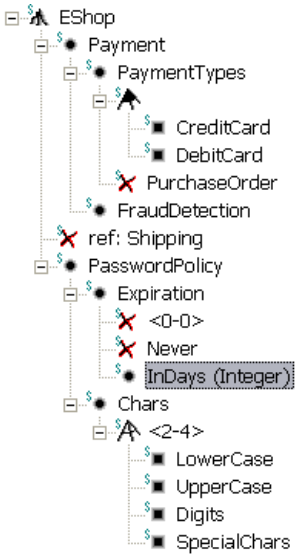

Figure 5: Example of a specialization of `EShop`

5. SPECIALIZATION OF FEATURE DIAGRAMS

Specialization of a feature diagram yields another feature diagram, such that the set of configurations denoted by the latter diagram is a subset of the configurations denoted by the former diagram [10]. Specialization is useful if configuration needs to be performed in stages. For example, a diagram describing a security profile offered by the computing infrastructure of an organization could be specialized for each department of the organization, and then for each computer within the departments. Another application is when an organization wants to provide specialized product lines to different customers based on one common, larger product line, which it maintains internally (see [11] for concrete examples).

When specializing a diagram, a copy of the original diagram is created and series of specialization operations can be applied to the copy, such as refining feature and group cardinalities, removing and selecting features from a group, assigning attribute values, cloning solitary features, and unfolding references to feature diagrams (see [10] for a precise definition of the specialization operations). The resulting diagram is guaranteed to be a specialization of the original one. Figure 5 shows a specialization of our `EShop` feature diagram. The specialization operations are available from the context menu of a selected feature, group, or reference. In our example in Figure 5, the references to `Payment` and `PasswordPolicy` were unfolded, `PurchaseOrder` and `Never` were removed from their respective groups, `InDays` was selected from its group, the cardinality of `ref:Shipping` was refined to [0..0], and the cardinality of `FraudDetection` was refined to [1..1]. Note that empty groups and removed features are marked by the cross symbol ✗ and still visible, but their display could also be suppressed. Of course, a specialized diagram can be used as a basis for configuration or another specialization.

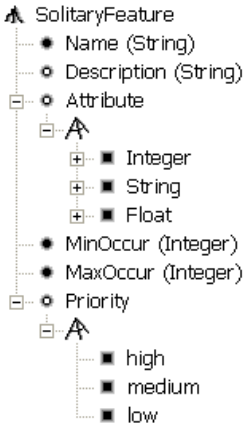

Figure 6: Metamodel for the concept of a solitary feature

6. USER-DEFINED ANNOTATIONS

During feature modeling, in addition to feature diagrams, other information, such as examples of existing systems implementing a given feature, feature priorities, stakeholders interested in a given feature, binding times, implementation status, developers responsible for a given feature, etc., needs to be recorded as annotations. The annotations may need to be attached to the feature model itself and/or its elements, such as diagrams, features, groups, and references. However, exactly what information needs to be recorded is usually project-dependent. This requirement can be accommodated by providing a user-extensible metamodel of the feature notation [2]. The metamodel is represented as a feature model itself and contains definitions of feature diagrams, features, groups, references, etc. Figure 6 shows the metamodel for the concept of a solitary feature, which has been extended with the user-defined feature `Priority` and its subfeatures. The other features in the metamodel are system-defined. When editing a feature model, the information associated with a selected element is shown in a properties view, which displays a configuration of the corresponding metamodel. This is demonstrated in Figure 7, which shows a fragment of our sample feature model and the information associated with the selected feature `FlatRate`. The user can access the metamodel of a given feature model through the Meta-Modeling tab shown in Figure 7 and edit it just as any other feature model, except that the system-defined features in the metamodel cannot be removed or renamed.

7. RELATED WORK

AmiEddi [22, 18] was the first editor supporting the feature modeling notation from [9]. That notation did not have feature and group cardinalities. As a successor to AmiEddi, CaptainFeature [2, 3] implements a cardinality-based notation that is similar to the one described in this paper. However, CaptainFeature renders feature diagrams in the FODA-style with extensions from [9, 10] (see Figure 8), whereas FeaturePlugin uses the tree widget of the Standard Widget Toolkit (SWT).

ConfigEditor [8] was an initial prototype implementing feature-based configuration. This functionality was later integrated into CaptainFeature. The configuration interface

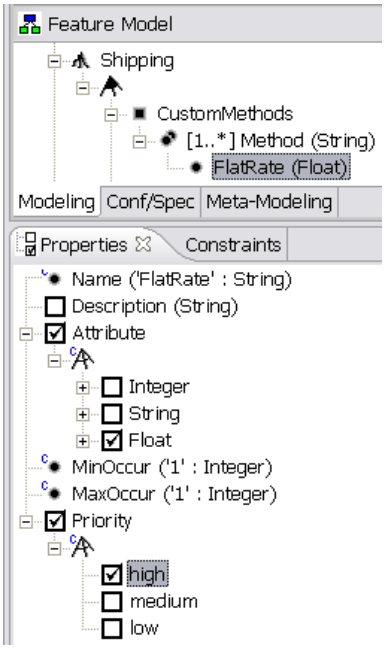

Figure 7: Example of specifying the properties of a solitary feature

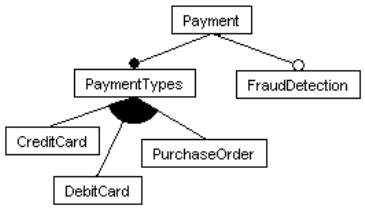

Figure 8: Alternative rendering style for feature models

described in this paper removes the strict top-down configuration order of ConfigEditor and CaptainFeature.

ReqiLine is a research tool integrating feature modeling and requirements engineering [23]. The tool can render feature models similar to CaptainFeature. The actual feature notation does not support cardinalities, but allows defining various relationships between features across the feature hierarchy. The tool can check models for consistency and has basic product configuration capabilities.

Pure::Variants is a commercial tool supporting feature modeling and configuration using a tree-view rendering similar to the one presented in this paper, but without feature cardinalities [20, 4]. In other words, Pure::Variants does not support cloning. Pure::Variants allows modeling global constraints between features and it offers interactive, constraint-based configuration using a Prolog-based constraint solver. Although not directly based on feature modeling, GEARS is another commercial tool for modeling and configuring software variants [15]. GEARS models variability as sets of parameters, where different parameter types stand for different kinds of variability, e.g., Boolean for optionality, enumeration for alternatives, and set for taking subsets. Although the parameters are not arranged into hierarchies as in feature diagrams, they can be organized

into separate modules related by import statements. Global constraints among parameters can also be defined and they will be checked during product configuration, which is done through interactive forms. Both Pure::Variants and GEARS have a built-in model of configuring sets of actual software components such as source files.

Another class of software configuration tools is emerging based on configuration research in Artificial Intelligence (AI), which in the past has been mostly focusing on configuring physical products [21]. Recent examples of applying AI configuration techniques to software component and product-line configuration are works by Myllärniemi et al [19], MacGregor [17], and Krebs et al [14]. The main difference between AI configuration approaches and configuration approaches based on feature modeling is the richness of the underlying structure. Most AI configuration approaches are based on an object model with classification (is-a) and aggregation (part-of) and user-defined relationships, over which local and global constraints are defined [21]. Feature modeling is focused on capturing choices (e.g., alternative and optional features) rather than different kinds of relationships, which is left to other modeling notations such as UML.

An approach to feature modeling and configuration based on XML was proposed by Cechticky et al. [5]. In this approach, a feature model is entered directly as XML using an XML editor guided by an XML Schema representing the metamodel of the feature modeling notation. Similarly, a configuration is entered as XML using another XML Schema that was generated by an XSLT script from the XML representing the feature model. Another XSLT script is run on the configuration to detect any constraint violations. Although this approach goes a long way and may be the preferred one in settings where XML is already in heavy use, dedicated tools for feature-model editing and configuration can provide more assistance to the user.

As of writing, none of the tools listed above supports feature-diagram specialization.

8. DISCUSSION AND FUTURE WORK

The basic structure of FeaturePlugin was generated from an EMF model of the feature modeling notation. This saved us a significant amount of development effort. The generated editor plug-in was customized to provide the necessary views and operations. Both the EMF model and the additional customization code were developed in an iterative and incremental way. Some requirements with respect to the user interface were not easily accommodated by the provided infrastructure and required some workarounds or slight compromises. However, the development effort savings due to infrastructure reuse outweighed these difficulties by far.

The user interface for rendering feature models as described in this paper allows for very efficient entry and editing of large feature diagrams. The current FeaturePlugin interface supports entering and editing feature diagrams using the keyboard only. Feature names and attribute values can be edited in-place without the need to switch to the properties view. Thanks to the ability to collapse the children of a node, large diagrams can be browsed and edited without the need to split them into smaller parts using feature diagram references.

The traditional notation shown in Figure 8 is often useful when presenting or discussing feature diagrams, but it only works well when the diagrams are relatively small. However, this problem can be addressed by providing filtering and selection facilities to show only the currently relevant part of a larger feature model. We plan to offer this notation as an alternative view, which will be developed using the Graphical Editor Framework of Eclipse. We also plan to provide a rendering using a table-tree view, in which selected additional information associated with each feature can be shown in a tabular form. This view will be useful during product-line scoping to display feature valuations from different stakeholders in a compact form.

In our future work, we plan to gain more experience with applying FeaturePlugin to drive template-based code generation (similar to [8]), configuring models, and providing runtime configuration in a variety of application domains. Among others, we plan to explore using feature models to specify different kinds of editors to be generated within the EMF infrastructure and to configure aspects in the AspectJ plug-in. Further development of FeaturePlugin will include additional configuration interfaces (e.g., forms) and renderings of feature models. Also, we plan to allow customizing the order in which the configuration wizard traverses a feature diagram and the content of each wizard page through appropriate diagram annotations.

FeaturePlugin is available at `http:\\gp.uwaterloo.ca\fmp`.

9. ACKNOWLEDGMENTS

The authors would like to thank Dr. Simon Helsen for his invaluable input during numerous design meetings and his extensive feedback as a user. Further thanks go to Sean Lau for his comments on an earlier draft, to Krzysztof Pietroszek for his effort in developing the wizard-based configuration, and to Peter Kim and Krzysztof Pietroszek for developing the support for additional constraints.

10. ABOUT THE AUTHORS

Michał Antkiewicz is a PhD candidate in the Department of Electrical & Computer Engineering (ECE), University of Waterloo. Krzysztof Czarnecki is an Assistant Professor in the Department of ECE, University of Waterloo, where he leads a research group on generative and model-based software engineering.

11. REFERENCES

[1] M. Barbeau and F. Bordeleau. A protocol stack development tool using generative programming. In D. Batory, C. Consel, and W. Taha, editors, *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02), Pittsburgh, October 6-8, 2002*, LNCS 2487, pages 93–109. Springer-Verlag, 2002.

[2] T. Bednasch. Konzept und Implementierung eines konfigurierbaren Metamodells für die Merkmalmodellierung. Diplomarbeit, Fachbereich Informatik, Fachhochschule Kaiserslautern, Standort Zweibrücken, Germany, Oct. 2002. Available from `http://www.informatik.fh-kl.de/~eisenecker/studentwork/dt_bednasch.pdf` (in German).

[3] T. Bednasch, C. Endler, and M. Lang. CaptainFeature, 2002-2004. Tool available on

SourceForge at https://sourceforge.net/projects/captainfeature/.

[4] D. Beuche. *Composition and Construction of Embedded Software Families*. PhD thesis, Otto-von-Guericke-Universität Magdeburg, Germany, Dec. 2003. Available from http://www-ivs.cs.uni-magdeburg.de/~danilo.

[5] V. Cechticky, A. Pasetti, O. Rohlik, and W. Schaufelberger. XML-Based Feature Modelling. In J. Bosch and C. Krueger, editors, *Software Reuse: Methods, Techniques and Tools: 8th International Conference, ICSR 2004, Madrid, Spain, July 5-9, 2009. Proceedings*, volume 3107 of *Lecture Notes in Computer Science*, pages 101–114. Springer-Verlag, 2004.

[6] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[7] K. Czarnecki. Overview of Generative Software Development. In *Proceedings of the European Commission and US National Science Foundation Strategic Research Workshop on Unconventional Programming Paradigms, September, 15–17, 2004, Mont Saint-Michel, France*, 2004. http://www.swen.uwaterloo.ca/~kczarnec/gsdoverview.pdf.

[8] K. Czarnecki, T. Bednasch, P. Unger, and U. W. Eisenecker. Generative programming for embedded software: An industrial experience report. In D. Batory, C. Consel, and W. Taha, editors, *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02), Pittsburgh, October 6-8, 2002*, LNCS 2487, pages 156–172. Springer-Verlag, 2002.

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

[10] K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process Improvement and Practice*, 10(1):7–29, jan/mar 2005. http://swen.uwaterloo.ca/~kczarnec/spip05a.pdf.

[11] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration through specialization and multi-level configuration of feature models. *Software Process Improvement and Practice*, 10(2), 2005. http://swen.uwaterloo.ca/~kczarnec/spip05b.pdf.

[12] M. Griss, J. Favaro, and M. d' Alessandro. Integrating feature modeling with the RSEB. In *Proceedings of the Fifth International Conference on Software Reuse (ICSR)*, pages 76–85. IEEE Computer Society Press, 1998.

[13] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90TR -21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Nov. 1990.

[14] T. Krebs and L. Hotz. Needed expresiveness for representing features and customer requirements. In *Proceedings of Modelling Variability for Object-Oriented Product Lines (ECOOP 2003 Workshop), Darmstadt, Germany, July 21 2003*, July 2003. Available from http://lki-www.informatik.uni-hamburg.de/~krebs/publications/ECOOP2003.%pdf.

[15] C. W. Krueger. Software mass customization. White paper. Available from http://www.biglever.com/papers/BigLeverMassCustomization.pdf, Oct. 2001.

[16] K. Lee, K. C. Kang, and J. Lee. Concepts and guidelines of feature modeling for product line software engineering. In C. Gacek, editor, *Software Reuse: Methods, Techniques, and Tools: Proceedings of the Seventh Reuse Conference (ICSR7), Austin, USA, Apr.15-19, 2002*, LNCS 2319, pages 62–77. Springer-Verlag, 2002.

[17] J. MacGregor. Expressing domain variability for configuration – invited paper. In F. Oquendo, B. Warboys, and R. Morrison, editors, *Software Architecture: First European Workshop, EWSA 2004, St Andrews, UK, May 21-22, 2004. Proceedings*, volume 3047 of *Lecture Notes in Computer Science*, pages 230–240. Springer-Verlag, 2004.

[18] Mario Selbig. AmiEddi, 2000-2004. Tool available at http://www.generative-programming.org.

[19] V. Myllärniemi, T. Asikainen, T. Männistö, and T. Soininen. Tool for configuring product individuals from configurable software product families. In T. Männistö and J. Bosch, editors, *Proceedings SPLC 2004 Workshop on Software Variability Management for Product Derivation – Towards Tool Support*, pages 24–34. Technical Report 6 – HUT-SoberIT-C6, Aug. 2004. Available from http://www.soberit.hut.fi/SPLC-DWS/.

[20] pure-systems GmbH. Variant Management with Pure::Consul. Technical White Paper. Available from http://web.pure-systems.com, 2003.

[21] D. Sabin and R. Weigel. Product configuration frameworks — a survey. *IEEE Intelligent Systems*, 13(4):42–49, 1998.

[22] M. Selbig. A feature diagram editor — analysis, design, and implementation of its core functionality. Diplomarbeit, Fachbereich Informatik, Fachhochschule Kaiserslautern, Standort Zweibrücken, Germany, Oct. 2000.

[23] T. von der Maßen and H. Lichter. RequiLine: A Requirements Engineering Tool for Software Product Lines. In F. van der Linden, editor, *Software Product-Family Engineering: 5th International Workshop, PFE 2003, Siena, Italy, November 4-6, 2003. Revised Papers*, volume 3014, pages 168–180, 2004.

[24] D. M. Weiss and C. T. R. Lai. *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, 1999.

[25] World Wide Web Consortium. *XML Path Language (XPath) 2.0*, 2005. http://www.w3.org/TR/xpath20/.