Integrated Product Line Model for Semi-Automated Product Derivation Using Non-Functional Properties

Norbert Siegmund, Martin Kuhlemann, Marko Rosenmüller, Christian Kaestner, and Gunter Saake
University of Magdeburg
39106 Magdeburg, Germany
{nsiegmun,mkuhlema,rosenmue,ckaestne,saake}@ovgu.de

Abstract

Software product lines (SPLs) allow to generate tailor-made software products by selecting and composing reusable code units. However, SPLs with hundreds of features and millions of possible products require an appropriate support for semi-automated product derivation. We envision this derivation to be extended by non-functional properties that are associated to code units and domain features. Code units and domain features are commonly organized in different models and connected via complex mappings, what make automation difficult. We propose a model that integrates features and code units in order to allow semi-automated product derivation using non-functional properties.

1 Introduction

Software product lines (SPLs) aim at providing variability for a family of similar software products tailored to individual user needs [12]. Variation points of an SPL, i.e., the functional differences between different product line members [4], are analyzed and modeled during domain analysis as features inside a *feature model* [20, 14]. Based on feature models SPLs are implemented using reusable and modular *code units* that are organized in an *implementation model*. Product line members are *derived* by composing such code units.

In small SPLs, it is usually simple to derive a product by selecting the required features manually. However, as the size of SPLs grows – large SPLs in industry may contain over 1000 features [31, 25] – the derivation process of selecting these features becomes more tedious and difficult, because many decisions are necessary, each requiring knowledge of the SPL's domain and maybe of implementation.

Product derivation becomes further complex in the presence of non-functional constraints, e.g., in domains like embedded systems where resources are restricted. There are many relevant non-functional constraints [19], for example, a generated database management product should have a maximum footprint size of 48 KB to fit on an embedded device and must be capable of handling a throughput of 10 transactions per second (T/s) because input is provided at this rate. To derive a product by configuring hundreds of variation points, that additionally has to adhere to non-functional constraints is difficult and often results in a trial-and-error approach, which is tedious and error-prone.

We envision tool support that assists developers in selecting features to support the product derivation process. For example, tools can automatically hide variation points that are irrelevant because of constraints and features selected earlier inside configuration process. In the following, we refer to this process as *semi-automated derivation (SAD)* [32, 5, 36]. We argue that SAD is particularly promising in the presence of non-functional constraints. For example, tool support could check which features cannot be selected because they would violate a footprint constraint.

SAD tools require domain specific information about the SPL, that come solely from the feature model in existing approaches (i.e., features and constraints between features). However, to define non-functional constraints we need additional information. While some non-functional properties, like development time, can be directly attached to the feature model [5], others, like performance, binary code size, and in-memory size, depend on the implementation and can be associated with code units [36]. Therefore, we have to consider both, feature model and implementation model, for SAD.

Current approaches to SPL development typically use a mapping between feature model and implementation model which makes SAD with non-functional constraints difficult because the intermediate result of selecting code units using non-functional properties in the implementation model must be propagated back to the feature model used for configuration. In this paper, we suggest an *integrated software*

product line model (ISPLM), that combines both, feature model and implementation model, to overcome problems in the SAD process with two models. This model should provide the basis for creating an SAD tool that supports the user in deriving products based on non-functional constraints. In our long term vision, this model enables an adequate handling of large and complex SPLs in resource constrained environments.

2. Background

In this section, we give an overview of feature modeling and current approaches for implementing and configuring SPLs.

Feature Modeling. *Feature-oriented domain analysis* (FODA) [20] is the process of identifying and collecting information relevant for a stakeholder that describe the features of a concrete domain. These features might be modeled with additional information like attributes or annotations [14] and are integrated into a feature model with further domain constraints. Features can be mandatory or optional and may have relations or constraints to other features, e.g., two features can be alternative. The feature model is typically visualized by a feature diagram that is a hierarchical representation of all features of an SPL.

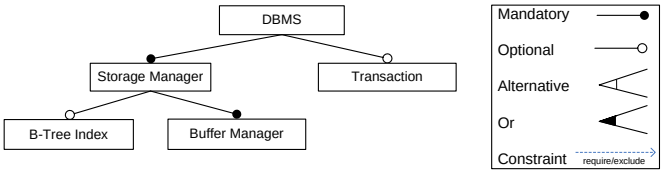

Figure 1. Simple Feature Diagram.

Figure 1 depicts a sample feature diagram of an SPL for a *database management system (DBMS)*. The diagram consists of the base concept DBMS as the root node which represents the core functionality of the DBMS and additional nodes that represent features of the product line. The feature diagram shows that only feature *Storage Manager* is mandatory for every product because of the required storage functionality for every DBMS instance provided by this feature. Feature *B-Tree Index*, which represents a special data structure for accessing data in a DBMS, is optional, i.e., a stakeholder has to decide, whether this special feature should occur in a product. Further relations between features are possible, e.g., *excludes* and *implies*, but not shown in the Figure.

SPL Implementation. Code units implement the features of an SPL [14]. A common practice is the realization of code units using components [12]. A mapping assigns features to code units that implement the according functionality. In general, code units can implement multiple features and crosscutting features may map to several code units [8]. All code units and constraints between them form the implementation model (a.k.a. architecture model [23]).

An important difference between common SPLs and SPLs in the embedded systems domain are alternative implementations. Alternative implementations are required for fine-grained adjustments of non-functional properties by providing equivalent functionality. Figure 2 depicts two different implementations of a B-Tree feature. Component *B-Tree small* implements basic functionality and is optimized for binary code size at the costs of performance. In contrast, component *B-Tree fast* uses special algorithms, e.g., lazy deletion [37, 18], that increases performance at the costs of binary code size. Such a need for specialized algorithms is common in embedded systems [10, 35]. The *Buffer Manager* functionality (cf. Figure 1) provides support for different storage types. It similarly has two variants, one for a simple data handling without any specialized memory structures (*Minimal Buffer Manager*) and one for performance optimized data handling (*Unrestricted Buffer Manager*). The developer has to decide which code unit is optimal for a given environment. As shown in Figure 2 there are constraints between code units as well.

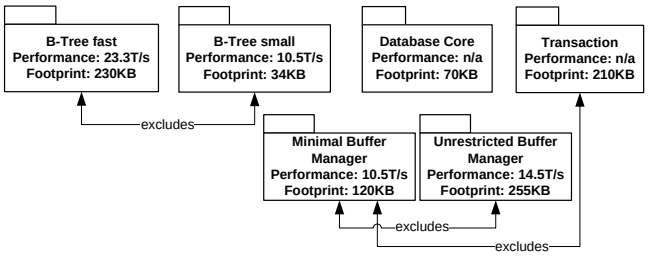

Figure 2. Implementation Model.

Product Derivation. To derive a product of an SPL, the stakeholder decides which features to include. Feature selection is usually realized based on a feature model [21, 1, 2, 27, 29]. During feature selection, tools can check the current configuration against existing constraints that are defined in the feature model and implementation model.

3 Semi-automated Derivation with Non-Functional Constraints

SAD is an approach to assist a user during configuration of a large SPL with many features. This support can be realized by automatically hiding features that cannot be selected at the current state of configuration due to existing

constraints [2, 15, 11, 7]. Other approaches guide the user through the configuration space and further visualize dependencies between features [27, 29]. Our vision goes beyond this derivation process based only on the feature model, i.e., we also want to include non-functional constraints that have to be fulfilled in the derived product.

We envision an extension to the concept of SAD by an automated selection of features and code units according to *non-functional requirements*. Often non-functional properties depend on how a code unit is implemented. Therefore, it should be possible to present hints to a stakeholder during configuration of an SPL which show how a selection affects the properties of the final software. To start the SAD process, a user defines constraints, e.g., *Footprint* $<$ *48KB AND Performance* $>$ *10T/s*, for the resulting software which may already exclude certain features from the configuration space. As a next step, the user selects needed functionality of the SPL. After every decision the SAD tool supports the user by giving hints or automatically selecting features or code units according to the constraints. Thus, the SAD process requires information from the feature model (e.g., features, domain constraints) as well as from the implementation model (e.g., non-functional properties of code units like binary code size and reliability). The *measurement* of such non-functional properties of code units is in the focus of our research, but outside the scope of this paper[1].

4 Problem Statement

In the following we present problems we found that result from the separation of feature model and implementation model.

SAD Tools. During our development of an SAD tool, we observed several problems. When evaluating a user selection, we have to proof this selection against domain constraints defined in the feature model. Afterwards, the tool has to map the current configuration to the implementation model. Again, we have to proof the same user selection of the feature, but now against the implementation model, because of implementation constraints, e.g., it has to be validated if *excludes* relations are violated. Moreover, additional requirements and constraints may result in an automatic selection of required code units that map back to a feature selection. This task is already complex, but the SAD tool, which supports non-functional constraints, has to evaluate the respectively actual configured implementation against the existing non-functional properties. These

[1] As part of the FAME-DBMS project (funded by the German Research Foundation, project no. SA 465/32-1), we work on the derivation of non-functional properties by composing products and measuring the resulting properties.

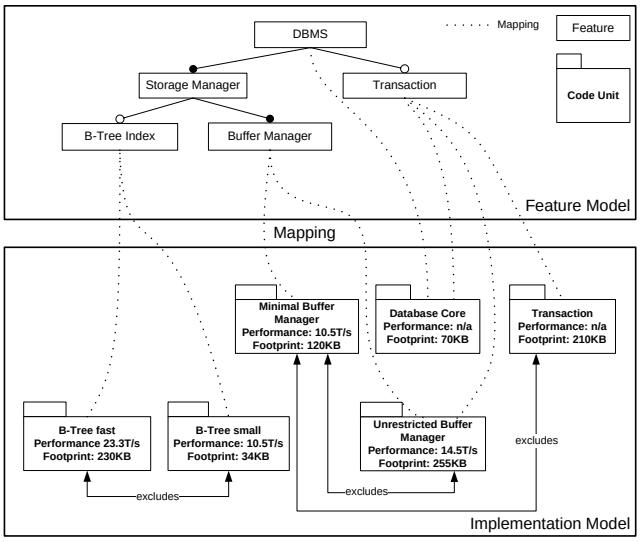

Figure 3. Problems of separated Models.

non-functional constraints can raise conflicts in the implementation selection and therefore, in the feature selection.

Figure 3 shows a simplified abstraction of a mapping between a feature model and an implementation model from our DBMS product line. If we define performance and footprint constraints and select the feature *Transaction*, an SAD tool has to map the selection to three different components. The tool has to check the *excludes* constraints of components *Transaction* and *Unrestricted Buffer* which results in a verification of the incomplete feature selection. Furthermore, the SAD tool has to check that the non-functional constraints are not violated. In this example, the configuration of component *Minimal Buffer Manager* might be changed to the selection of component *Unrestricted Buffer Manager* which leads to a change of the non-functional properties of the current configuration.

The reason for for this complex derivation process lies in a complex interaction of two separated models of *one* SPL that are typically connected via an intricate mapping [9, 34, 23] and have to be consistent. This complexity makes the development of SAD tools costly and time consuming and the SAD process expensive.

Interacting Code Units. Assigning non-functional properties to elements of one model can be difficult if these properties vary depending on the remaining module selection. Reasons for the changing values are mainly code interactions. The interaction code, a.k.a. *derivatives* [24] or *lifters* [26], arise if one feature crosscuts another feature. For example, the code units of *B-Tree fast* and *Unrestricted Buffer Manager* interact by including extra code if they occur in the same product. This is not shown in the diagram, because we use components as code units that usually in-

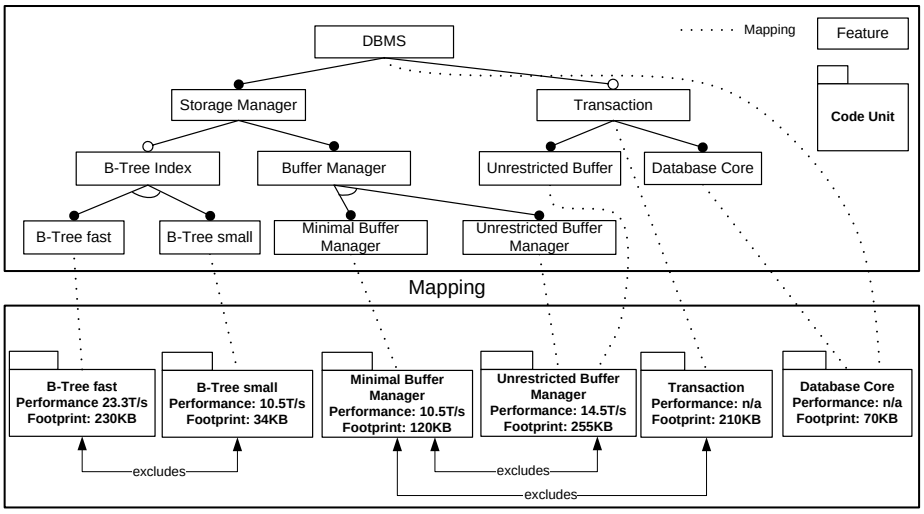

Figure 4. Redundant Representation of Code Units.

clude the whole interaction code so that it does not occur modularly. This additional code may lead to increased binary code size or affect performance. However, interaction code may change non-functional properties significantly, e.g., a *Transaction* interaction code can reduce the performance of the *Buffer Manager* by locking data in the memory. For an SAD tool, it is problematic to derive required non-functional properties if derivatives are not modeled explicitly. Because of the common integration of interaction code in one already existent code unit [22], it is not modeled separately in the implementation model. This results in a lack of expressiveness for the SAD process and therefore, it is a problematic investigation of the varying non-functional properties of interaction code.

Consistency. The problems of SAD tools and consistency described above, typically occur with a complex mapping. A possible solution for selecting alternative implementations during configuration, is the redundant representation of code units of the implementation model in the feature model. For example, the feature *B-Tree Index* could be modeled by two alternative subfeatures (cf. Figure 4) which represent the two alternative *B-Tree* components. This transformation, however, results in a mixture and duplication of both models which raises consistency problems and is error-prone. Additionally, SAD becomes more time consuming because it has to validate the code units twice. Considering changes in one model, like it is common during software evolution, the maintenance of the models becomes difficult. This is caused by the evolution of feature models which is separated from the evolution of implementation models and leads to an increasing mismatch between both models as already investigated by Tesanovic et al. [34].

5 Integrated Software Product Line Model

In the following we present our approach for an *integrated software product line model (ISPLM)*. In particular, we integrate code units into a feature model to improve SAD of an SPL.

5.1 Overview

In Figure 6 we show a meta model for our approach. The ISPLM of Figure 5 consists of one root feature, like feature *DBMS*. The feature *DBMS* has subfeatures that are connected with different relations, e.g., feature *Storage Manager* is mandatory and feature *Transaction* is optional. Our syntax for these constraints is equivalent to the FODA representation of the DBMS domain. We integrate code units into the feature model to represent the features' implementation, e.g., the code unit *Database Core* implements feature *DBMS*. Features and code units in the ISPLM can have non-functional properties as well as relations (*excludes* and *implies*).

The integration of code units into the feature model requires two conditions. First, code units can only be child elements of features or other code units. We do not allow to model a code unit as a root node or as a parent of a feature node because features are defined during the domain analysis which precedes the implementation phase (the feature model written once is solely extended but not changed). Second, we need an additional relation to represent the interaction between code units (i.e., derivatives, cf. Section 4) because of the interacting code units. The *Interaction* relation allows an SAD tool to automatically include the target code unit (filled rectangle) when all interaction sources are configured. In Figure 5, this is the case for the code unit

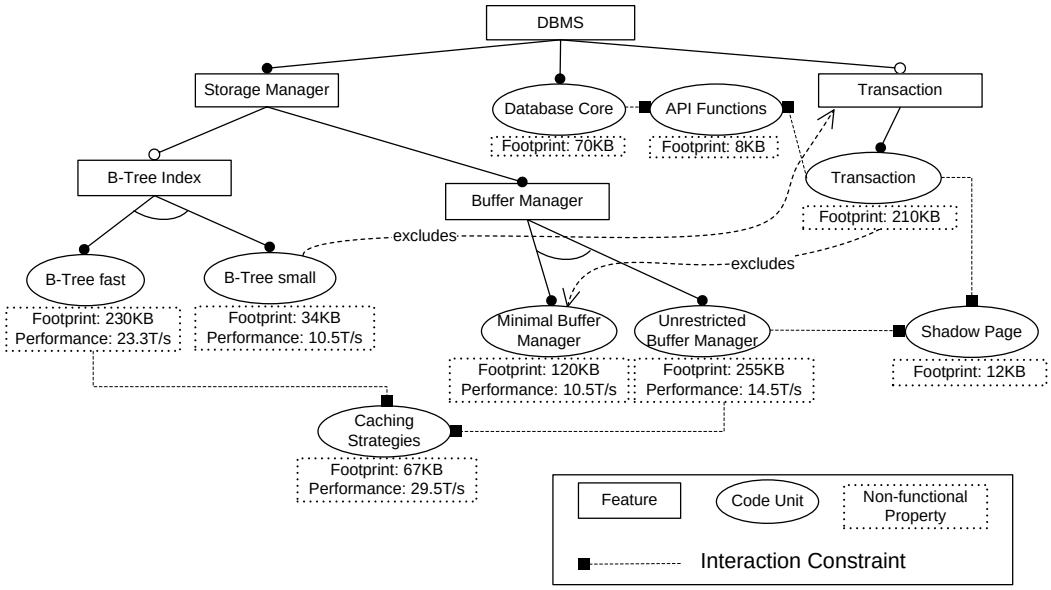

Figure 5. Diagram of the integrated Software Product Line Model.

Caching Strategies. The code unit and its non-functional properties influence the derivation process only if code unit *B-Tree fast* and *Unrestricted Buffer Manger* are selected. Relations can also exist between code units and features, e.g., consider the *excludes* relation between code unit *B-Tree small* and feature *Transaction*. With the ISPLM, it is possible to constrain the variation points of the domain space dependent on a selected code unit.

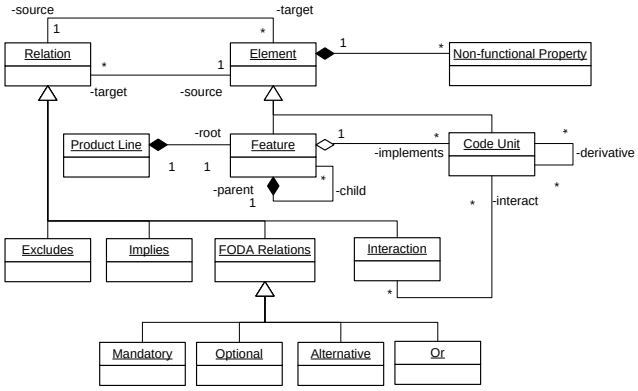

Figure 6. UML Metamodel of the SAD Model.

5.2 Benefits of ISPLM

Semi-automated Derivation. The integration of code units into the feature model allows to perform SAD without the effort of creating and maintaining two consistent models. It simplifies the implementation of SAD tools because information do not have to be propagated between both models. Moreover, representing interactions of code units inside the ISPLM provides an improved SAD because these interactions can be responsible for changes of non-functional properties.

User Benefits. The ISPLM allows a simplified visualization of non-functional properties in a diagram because no abstract and confusing mapping between feature model and implementation model is needed. We reason that this model can provide a basis for collaborating domain engineers and software engineers because the interaction between domain features and code units is apparent with one integrated model. We argue that maintaining one model is less error-prone than two different because both engineers use the same model which enhances the communication during software development. In particular, this additional information is needed for stakeholders who require implementation knowledge in deeply embedded systems and software engineers who require domain knowledge when implementing features.

Furthermore, the ISPLM explicitly allows to configure all variation points (features and code units) instead of only features.

Additional Benefits. Having an integrated model provides consistent changes in the domain as well as in the implementation. When restructuring the domain model the implementation is automatically adjusted, thus software evolution needs less effort. For instance, if the exemplified DBMS must run on an embedded device with feature *Transaction* but cannot fulfill the binary code size constraint, a

new code unit for feature *Transaction* might be needed. A software developer solely has to attach the new code unit to the feature *Transaction* and may define additional constraints to other code units. Additionally, maintenance costs for only one model instead of two independent models might decreased.

5.3 Discussion

The proposed model raises several discussion points, because it contradicts the well known separation of domain model and implementation model.

Separation of SPL Models. The separation of domain model and implementation model has the advantage of supporting independent implementation models by having a constant domain model. Ideally, the stakeholder and domain engineer should not need implementation knowledge nor be restricted by implementation issues. This distinction already blurs during the implementation phase where programming depends on domain modeling. Furthermore, this strict separation cannot be held up because the stakeholder obviously is interested in implementation dependent information, he at least wants to choose out of different implementations. In fact, embedded systems' SPLs need to mix the domain requirements with implementation requirements. Moreover, source code structures (e.g., variables, classes, etc.) could be directly generated from the domain model and their values are set up in the product derivation phase (e.g., configured numeric values). We argue that our SAD model is an appropriate model in areas where implementation requirements are relevant for a stakeholder.

Model Complexity. The integration of two models may increase complexity and size of the ISPLM compared to feature model and implementation model. Large SPLs with hundreds of features and a similar number of code units degrade usability of the whole model. To handle this problem we recommend views that filter only needed information. For example, one view could only represent the features and their constraints (the common feature model) and another view could show all code units including their relations (implementation model). In contrast to separated models the implementation view could contain parts of the domain model that are needed to understand the implementation. We enable the support of views by identifying the source and the target of relations and restrict the position of code units and features (e.g., a feature can only occur as a child of another feature).

6 Related Work

Several researchers aim at simplifying product derivation using SAD [2, 1, 15, 11, 27, 29]. Some researchers also include non-functional constraints for automated reasoning in extended feature modules. A prominent example is the work of Benavides et al. [5, 3, 6, 7], where non-functional properties, e.g., costs of a feature or its development time, are assigned to features. Automated product derivation strongly relates to the well known *constraint satisfaction problem*. This approach laid the basis for our work on SAD.

The next step for SAD is to include non-functional properties of product line code units. White et al. [36] published the tool *Scatter* that integrates non-functional properties into the product derivation process. In particular, *Scatter* includes the binary size of non-code data files (pictures). Products are derived using an extended constraint satisfaction solver presented in [5]. White et al. also investigated that code units can be modeled similar to feature models in the product line architecture model they proposed. In contrast to this approach, we go further to allow SAD with any kind of non-functional property that is related to code units of the SPL. Scatter handle only non-code data files. We evaluate code units of the resulting product instance. Moreover, we enable to represent changing properties of interacting code units (cf. Section 5.1).

Feature modeling gains much attention in recent research. Different feature models and extensions have been proposed, typically for tree-like diagram representation [21, 15, 16], in UML [17, 13], or using the object constraint language [30, 33] to improve domain modeling and domain reasoning, e.g., by adding cardinality and attributes to features. Extensions with attributes can also represent non-functional properties. However, our vision goes beyond just domain modeling and includes code units and their properties (potentially derived automatically) therefore we need an integrated software product line model.

A closely related approach by Reiser et al. [28] formalizes a unified feature model that includes features and code units in the same model. They argue that the heterogeneity of the SPL development process, methods, and tools is difficult to manage. They propose a framework to model artifacts and code units of a product line as an *artifact product line*. In other words the global product line consists of many small product lines which can be further decomposed in even smaller product lines. This approach models the implementation and makes the implementation selectable by configuring these small artifact product lines. However, this framework does not consider any non-functional properties, hence it does not allow the SAD process using non-functional properties.

Ziadi et al. [38] propose the use of UML to derive

products. They translate feature models into an equivalent UML product line architecture model. Based on this architecture model they allow the configuration of product instances. In contrast to our approach they do not consider non-functional properties neither the interactions of code units.

7 Conclusion and Further Work

In this paper we outlined our vision of *semi-automated derivation (SAD)* using non-functional properties of SPL products and discussed difficulties caused by the typical distinction between feature model and implementation model. Product derivation is a complex task if an SPL has hundreds of features and the implementation of features varies. To configure such an SPL with many variation points we propose to use non-functional constraints for supporting a user in product derivation.

We have shown that traditional approaches of modeling domain and implementation separately are insufficient. They do not consider alternative implementations of one feature nor non-functional properties completely. SAD tools have to validate the feature model and the implementation model to enable a configuration which is not always possible because of complex mappings between both models. In contrast, we presented a new *integrated software product line model (ISPLM)* that integrates code units and their non-functional properties into the feature model. We argue that the ISPLM reduces the effort for the SAD process and it improves consistency and evolution management.

In further work, we will continue to develop an SAD tool and implement the ISPLM. This will allow us to analyze and evaluate the resulting implementation effort and benefits in contrast to existing models in a case study.

Acknowledgments

Norbert Siegmund and Marko Rosenmüller are funded by German Research Foundation (DFG), Project SA 465/32-1. The presented work is part of the FAME-DBMS project[2] a cooperation of Universities of Dortmund, Erlangen-Nuremberg, Magdeburg, and Passau funded by DFG.

References

[1] M. Antkiewicz and K. Czarnecki. FeaturePlugin: feature modeling plug-in for Eclipse. In *Proceedings of the 2004 OOPSLA Workshop on Eclipse Technology eXchange*, pages 67–72. ACM Press, 2004.

[2] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the International Software Product Line Conference (SPLC)*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer, 2005.

[3] D. Batory, D. Benavides, and A. Ruiz-Cortes. Automated Analysis of Feature Models: Challenges Ahead. *Communications of the ACM (CACM)*, 49(12):45–47, 2006.

[4] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering (TSE)*, 30(6):355–371, 2004.

[5] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Automated Reasoning on Feature Models. *Advanced Information Systems Engineering: International Conference (CAiSE)*, 3520:491–503, 2005.

[6] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Corts. A first Step towards a Framework for the Automated Analysis of Feature Models. In *Managing Variability for Software Product Lines: Working With Variability Mechanisms*, pages 45–53, 2006.

[7] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Corts. FAMA: Tooling a Framework for the Automated Analysis of Feature Models. In *Proceeding of the First International Workshop on Variability Modelling of Software-intensive Systems (VAMOS)*, 2007.

[8] D. Beuche. pure::variants Eclipse Plugin. User Guide., 2004.

[9] D. Beuche, H. Papajewski, and W. Schröder-Preikschat. Variability Management with Feature Models. *Science of Computer Programming*, 53(3):333–352, 2004.

[10] C. Bobineau, L. Bouganim, P. Pucheral, and P. Valduriez. PicoDBMS: Scaling Down Database Techniques for the Smartcard. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 11–20, 2000.

[11] G. Botterweck, D. Nestor, A. Preuer, C. Cawley, and S. Thiel. Towards Supporting Feature Configuration by Interactive Visualization. In *International Workshop on Visualisation in Software Product Line Engineering (ViSPLE)*, pages 125–131, 2007.

[12] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[13] L. M. Cysneiros and J. C. S. do Prado Leite. Nonfunctional requirements: From elicitation to conceptual models. *IEEE Transactions on Software Engineering (TSE)*, 30(5):328–350, 2004.

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

[15] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Staged configuration using feature models. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 266–283, 2004.

[16] K. Czarnecki, C. H. P. Kim, and K. T. Kalleberg. Feature Models are Views on Ontologies. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 41–51, 2006.

[17] H. Gomaa. Designing Software Product Lines with UML. Addison-Wesley, 2004.

[18] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[2] http://wwwiti.cs.uni-magdeburg.de/iti_db/research/FAME

[19] International Organization for Standardization (ISO). ISO 9126 Software engineering – Product quality. ISO/IEC 9126-0, 2006.

[20] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[21] 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 (ASE)*, 5:143–168, 1998.

[22] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, 1997.

[23] U. Kulesza, V. Alves, A. Garcia, A. C. Neto, E. Cirilo, C. Lucena, and P. Borba. Mapping Features to Aspects: A model-based generative Approach. In *Workshop On Early Aspects (EA)*, 2007.

[24] J. Liu, D. Batory, and C. Lengauer. Feature-Oriented Refactoring of Legacy Applications. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 112–121. ACM Press, 2006.

[25] F. Loesch and E. Ploedereder. Optimization of Variability in Software Product Lines. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 161–160. IEEE Computer Society, 2007.

[26] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443. Springer, 1997.

[27] R. Rabiser, D. Dhungana, and P. Grnbacher. Tool Support for Product Derivation in Large-Scale Product Lines: A Wizard-based Approach. In *International Workshop on Visualisation in Software Product Line Engineering (ViSPLE)*, pages 119–124, 2007.

[28] M.-O. Reiser, R. Tavakoli, and M. Weber. Unified Feature Modeling as a Basis for Managing Complex System Families. In *Proceeding of the First International Workshop on Variability Modelling of Software-intensive Systems (VAMOS)*, pages 79–86, 2007.

[29] D. Sellier and M. Mannion. Visualizing Product Line Requirement Selection Decision. In *International Workshop on Visualisation in Software Product Line Engineering (ViSPLE)*, pages 109–118, 2007.

[30] P. Sochos, I. Philippow, and M. Riebisch. Feature-Oriented Development of Software Product Lines: Mapping Feature Models to the Architecture. In *Net.ObjectDays*, pages 138–152, 2004.

[31] M. Steger, C. Tischer, B. Boss, A. Müller, O. Pertler, W. Stolz, and S. Ferber. Introducing PLA at Bosch Gasoline Systems: Experiences and Practices. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 34–50, 2004.

[32] D. Streitferdt, M. Riebisch, and I. Philippow. Details of Formalized Relations in Feature Models Using OCL. *Engineering of Computer-Based Systems (ECBS)*, 00:297–304, 2003.

[33] D. Streitferdt, P. Sochos, C. Heller, and I. Philippow. Configuring Embedded System Families Using Feature Models. In *NetObjectsDay*, 2005.

[34] A. Tesanovic and M. de Jonge. Exploring Effects of Feature Mismatch to Evolution of Product Lines with Components and Aspects. In *GPCE Workshop on Aspect-Oriented Product Line Engineering (AOPLE)*, 2007.

[35] R. Vingralek. GnatDb: A Small-Footprint, Secure Database System. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 884–893, 2002.

[36] J. White, D. C. Schmidt, E. Wuchner, and A. Nechypurenko. Automating Product-Line Variant Selection for Mobile Devices. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 129–140, 2007.

[37] B. Zhang and M. Hsu. Unsafe operation in B-trees. *Acta Informatica*, 26(5):421–438, 1989.

[38] Ziadi, Tewfik and Jézéquel, Jean-Marc and Fondement, Frédéric. Product Line Derivation with UML. In *Proceedings Software Variability Management Workshop (SVM)*, 2003.