Product Derivation for Solution-Driven Product Line Engineering

Christoph Elsner†, Daniel Lohmann‡, Wolfgang Schröder-Preikschat‡
†Siemens AG, Corporate Technology & Research
‡Friedrich-Alexander University Erlangen-Nuremberg
†christoph.elsner.ext@siemens.com, ‡{lohmann,wosch}@cs.fau.de

ABSTRACT

Solution-driven product line engineering is a project business where products are created for each customer individually. Although reuse of results from former projects is widely done, configuration and integration of the results currently is often a manual, time-consuming, and error-prone task and needs considerable knowledge about implementation details.

In this paper, we elaborate and approach the challenges when giving automated support for product derivation (i.e., product configuration and generation) in a large-scale solution-driven product line context. Our PLiC approach resembles the fact that, in practice, the domain of a large product line is divided into sub-domains. A PLiC (product line component) packages all results (configuration, generation, and implementation assets) of a sub-domain and offers interfaces for configuration and generation. With our approach we tackle the challenges of using multiple and different types of configuration models and text files, give support for automated product generation, and integrate feature modeling to support application engineering as an extensive development task.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Design

Keywords

Software Product Line Development, Solution-Driven Software Development, Feature Modeling

1. INTRODUCTION

In classical software product line engineering (SPLE), a software product line is defined as a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [12].

SPLE, however, is not only about *products* in the strict sense, which are usually created according to market needs before being offered to the customer. It covers the realm of software *solutions* as well, meaning that the customer has high influence on the requirements of the software product to develop, which is then created in context of a customer-specific project leveraging reuse.

Feature modeling can guide the whole process of solution-driven product line engineering. In fact, Siemens already uses feature modeling to describe the problem space variability of possible products [18]. It supports scoping of the product line, tendering, cost planning, supports communication between sales&marketing and development, and helps scheduling development releases, testing, and evolution. For documentation purposes, requirements trace into features which in turn trace to the solution space assets. We describe this in further detail in [18].

Although this is far more than using feature models "as a sketch" only, there still remains a gap between the problem space, modeled with features, and the solution space. Several causes currently still hinder using the feature model for product configuration and support partly-automated product generation from the configuration. In particular for solution-driven product lines, where addressing customer-specific requirements and manual implementaion play a central role, an integrated concept for combining automated and manual product derivation is missing.

In this paper, we will (1) describe the characteristics of solution-driven product line engineering and then what we see as state-of-the-art of problem space feature modeling in industry. We (2) derive the challenges for using this feature model for product derivation (i.e., product configuration and generation) in a solution-driven context. We (3) propose and discuss the PLiC approach to tackle the challenges.

2. SOLUTION BUSINESS

Solution-driven business is a project business where results are created for individual customers and their specific problems. It is not the products fitting into a market segment or the optimized production process that is sold to customers. The result, this is the solution or "product" of such projects, is typically only sold once in this form. Examples of solution business are engineering of power plants, production lines, smart homes, hospitals and their building automation, and railway control centers.

FOSD'09, October 6, 2009, Denver, Colorado, USA.

Product Business	Solution Business
n defined products, some explicitly excluded	project business
configuration of fixed set of functions	no product is used twice
foreseeable variability	unanticipated variability
facilitate maintainability of PL platform and products	facilitate/replace copy & paste
"closed world"	open collection of artifacts
example: digital camera	example: power plant, smart home

Table 1: Product vs. Solution Business

Solution business does not mean that every single feature of the result is crafted from scratch. In fact, an efficient solution business is urged to reuse and reapply existing know-how and results to stay competitive. In product business the scope of the product line is fixed by the variability the feature model describes. In solution business, in contrast, each new project reshapes the scope of the product line.

Table 1 compares product and solution business to characterize their differences. The scope of a product line in product business is well defined. It comprises a number of products with a set of common and individual functions. The core asset base is managed, even when application engineering adds functions or when extending the scope of the product line. In solution business, "products" are not clearly defined and show a high variability.

Results in solution business projects share certain characteristics that are determined by the domain. It is therefore useful to manage them together in a software product line. However the different sub-domains of the product line may be rather heterogeneous. They may be covered by standard software *products*, or products derived from other company-internal *product lines*, while others in turn might be developed *from scratch*.

Solution-driven product line engineering basically means that the primary focus is on application engineering, while domain engineering diminishes. In such a scenario, not only the reuse of existing results, but also *product-specific adaptations* and *reactive product line evolution*, that is, when that the concrete applications drive the development of the core product line assets, play very important roles.

3. PROBLEM SPACE FEATURE MODELING

Feature modeling was introduced in [11] as part of the domain analysis and domain modeling phase to systematically describe the common and variable features shared among the products of a product line. A feature model represents the features of a family of systems in the domain and the relationships between them [11]. A valid selection of features from a feature model is called a configuration. Usually, a feature model is considered to be located in the problem space; that is the scope of the analyst who does not care about solution details. This means that the concepts and relations described there are condensed so that they can be understood without detailed implementation knowledge. The actual architectural models and the implementation assets, in contrast, reside in the solution space.

Scoping, in turn, is an analysis activity in domain engineering to find the boundaries of the whole product line, its reusable sub-domains, and its assets [3, 16, 17]. Because there is no agreed definition, we define a sub-domain of a product line as a subpart of the overall product line domain, whereas there is a high cohesion in its problem space, solution space, and in company-internal organization. A sub-domain may be, for example, the operating system, data storage, middleware, GUI frameworks, user management, or domain-specific services. Scoping requires being able to assign business value to decisions on what should be in/out of the product line and carry those decisions on when deciding on what functionality should be built within reusable sub-domains and assets.

Rescoping is necessary to keep the scope of the product line optimized during its evolution. In solution business, rescoping is triggered (at least) for every new project to decide, if a necessary adaptation shall be developed reusable in the context of the overall product line. In this case it must be assigned to an appropriate sub-domain; else it is developed product-specific without following measures for later reuse.

Feature modeling and scoping is a good match. Features are a natural way to describe a domain in terms of problem space concepts [17]. They can be refined and related to solution space assets, and so can carry on business value information. A feature model can, hence, serve as scoping model and define the overall product portfolio and a strategic vision of the variability of the product line. The model is successively refined to map features to concrete sub-domains, and, finally, draw trace links into implementation components of a system. The scoping model includes all common and variable features of a product. The product line architect uses the scoping model to design the reference architecture for all the products.

Once feature modeling is established, it can support various other planning and management tasks. In [18] we describe how it has been used within Siemens to support project and iteration planning and controlling. In general, various task for bridging the "communication gap" between product management, sales&marketing, requirements management, architecting, and development may be supported by the data, like tendering, cost planning, and product line evolution.

In our view, the *essential requirement* on a problem space variability modeling language—and maybe the key success factor of feature modeling—lies in its easy understandability. Involved stakeholders from various backgrounds can immediately grasp the concepts and understand the semantics of feature diagrams, and, at least from the high-level point of view, the expressiveness of (cardinality-based) feature models [6] is sufficient.

However, the downside follows when the real-world solution is to be derived from the asset base. The problem space feature model is far from complete; it only contains high-

level concepts that are not sufficient for complete product configuration. Although there may exist traces from features to the sub-systems and components it affects, *how* to include, exclude, connect, or parameterize a component, in especially, how to generate the actual product, is not part of a problem space model.

Last but not least, variability modeling by only using feature models is quite limited. Constructive variability (instantiation, references) is better expressed in languages that support this concepts directly, and domain experts, for example for business processes, will prefer doing certain configuration tasks using domain-specific languages (DSLs) such as BPEL instead of (mis-)using feature modeling.

4. EXAMPLE SOLUTION-DRIVEN PRODUCT LINE

To illustrate the general problem and the solution proposed in this paper, let us consider the software for a medical digital assistant (MDA) product line for medical hospital personnel. A MDA is basically a personal digital assistant (PDA) with custom software connecting to the hospital information system. For simplification, we will consider two sub-domains: the embedded operating system including the middleware (OS), and the domain-specific business logic including the graphical front-end (GUI). The OS sub-domain is also used in other product lines (e.g., intelligent displays), whereas the GUI sub-domain is only of use in MDAs. The latter needs considerable configuration and manual implementation effort for each hospital, depending on the medical services and workflows it disposes of; equipping a hospital with MDAs is therefore solution business. Both sub-domains have a problem space feature model describing their high-level features. For planning purposes, the MDA product line itself is regarded as a sub-domain and comprises a problem space feature model, which is tied to the two other ones. The OS sub-domain implementation mostly reuses standard software and can be configured via separate text files, whereas the reusable parts of the GUI sub-domain have data and workfow models as configuration input.

5. CHALLENGES OF PRODUCT DERIVATION SUPPORT

Connecting the problem space variability model formally to solution space artifacts for product configuration and generation support is challenging. It requires an unambiguous translation of the concepts of both realms. We identified the following challenges, which, even though they also appear in a product-driven context to a certain extent, are of major importance for efficient product derivation for large-scale, solution-driven product line engineering:

1. Distributed Configuration
   The high-level feature model would become unmanageable if it contained all variability information for all sub-domains. Instead, there should be several variability models to divide and conquer the problem, similar to the hierarchical decomposition of sub-domains of the product line. Constraints between the variability models are necessary to enforce sub-domain–crossing dependencies. This distributed structure also goes in line with staged and multi-level configuration [7], where different stakeholders at different times are responsible for configuring different sub-domains.

2. Heterogeneous Configuration
   The high-level feature model constitutes our link from problem to solution space, and the detailed variability of some sub-domains might be described with feature models as well. However, there are also sub-domains where other forms of configuration are much more suited. Workflow or state machine models describe system behavior, other domain-specific (modeling) languages may describe deployment or replication. Finally, the basic infrastructure often bases on plain text configuration files. Constraints spanning different types of configuration must be possible.

3. Solution Generation Support
   Automating solution creation for those sub-domains that are mature enough to support generation requires connecting the configuration to implementation assets. Heterogeneous types of product generation (e.g., based on models and code generation, compilation, descriptor files, etc.) must be supported as well as a hierarchical mechanism to delegate generation call to all sub-domains.

4. Handling Application *Engineering*
   A new solution may require considerable effort for implementing new features. It is crucial that product-specific features are not neglected, but integrate neatly into the overall configuration and generation process.

Referring to the example in Section 4, the configuration of the MDA sub-domain is hierarchically distributed over the two sub-domains OS and GUI, which have heterogenous types of configuration (text files, models). Generative support only refers to separate sub-parts, and does not allow generating an overall MDA, and the problem space feature models are not used for actual product configuration. There is no sub-domain spanning constraint-checking for validating a configuration, and application engineering is not integrated into the sub-domain configuration and generation process.

6. APPROACHING THE CHALLENGES

In the following, we propose an approach to address the mentioned challenges. We will discuss it in Section 7.

6.1 Product Line Components

In practice, the domain of a product line is divided into sub-domains according to system boundaries and development responsibilities. Our approach encapsulates all artifacts related to a sub-domain—these are *configuration* artifacts, *generation* artifacts, and *implementation* artifacts—into one conceptual entity, a *product line component (PLiC)* (see Figure 1).

Since sub-domains can be hierarchically composed, this is also possible for PLiCs. A PLiC is a development and build-level entity, so interfacing of PLiCs works on the upper two layers: configuration and generation layer. PLiCs are a hierarchical concept, so each PLiC delegates requests for configuration and generation also to child PLiCs. It provides two interfaces: the *configuration interface* and the *generation interface*:

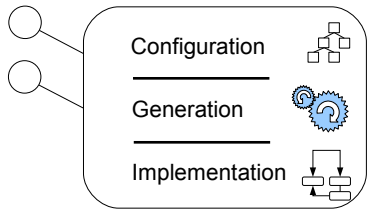

Figure 1: Product Line Component

- Configuration Interface
  - *Configure_PLiC*
    Configuration is a manual task and the interface therefore a human-machine interface. As, in practice, feature modeling alone is often not suited for overall product configuration, we expect that various domain-specific kinds of models or textual configuration languages become necessary. The top level PLiC will be configurable according to the problem space feature model, while the PLiCs for the sub-domains may have arbitrary domain-specific types of models for configuration. Note, that, as PLiCs are hierarchical, this provides support for hierarchical product lines [5], so that a whole (sub-)product line may be a part of the overall solution-driven product line.
  - *Check_Configuration*
    Checking a configuration requires evaluating constraints over all involved options. As these may spread multiple kinds of configuration models and textual files and also may cross sub-domains we need a suitable checking mechanism. Current modeling frameworks, such as EMF [10], fulfill this purpose. All domain-specific meta models developed with EMF correspond to the meta modeling infrastructure ECore. XText [22] facilitates rapid development of parsers for arbitrary textual languages. It outputs a corresponding EMF model for a file written in the language. For other types of models there already exist ECore converters (pure::variants [2] feature models, UML [14]) or can be developed as well. EMF's validation languages (OCL, oAW Check [13]) make building up a *constraint checking infrastructure* over several models and model types feasible.
- Generation Interface
  - *Generate_Solution*
    During application generation the configuration is evaluated and the product is built according to it. For stable sub-domains, the corresponding PLiC may encapsulate a so-called configurable product base [4] so that the product may be derived completely by using generative techniques. For the moment, we regard the generation facilities as a black box, so that arbitrary types of compilation and text and model-based generation and transformation techniques may be used internally.

By bundling configuration and generation facilities directly with the implementation assets, PLiCs approach Challenges 1 to 3. The configuration can be *distributed* over an arbitrary number of models and *heterogeneous* model types and *generation* is carried out hierarchically according to the PLiC hierarchy.

To sum up, our approach implies a hierarchy, where the root PLiC basically contains the problem space feature model to describe variability and global constraints. Referring to our example from Section 4 this would be the MDA problem space feature model and additional constraints. Enforcing correct configuration of the sub-domains conformant to this "abstract" feature model configuration is done via the constraint checking infrastructure. This ensures the configuration of all PLiCs (e.g., the sub-domains OS and GUI) to be valid. Finally, each PLiC generates a sub-product corresponding to its sub-domain, which makes product generation transparent regarding the concrete generation type (e.g., based on model transformations and code generation, pre-processors, etc.). In our example the OS PLiC generates the operating system and the middleware that specifically suite the needs of the domain-specific services and the graphical front-end generated by the GUI PLiC.

6.2 Supporting Solution-driven PLE

To approach Challenge 4, we have to distinguish different types of sub-domains. For solution-driven PLE, certain sub-domains of the solution may be derived from company-internal sub–product-lines, while others are covered by standard software, and others need manual implementation.

6.2.1 Sub–product-lines

Company-internal sub–product-lines fit very nicely into the overall concept. PLiCs have a hierarchical structure, so a hierarchical product line can be built. A sub–product-line PLiC exposes its feature model to the overall feature model. The constraint checking infrastructure ensures a globally valid configuration.

6.2.2 Standard Software

Incorporating standard software (e.g., for infrastructure) into the overall product line is also straight-forward. This means to encapsulate the standard software into a PLiC as well, using the same mechanisms as for sub–product-lines. This way, the detailed configuration of the standard software can be performed in the same manner as for company-internal sub–product-lines, and configuration constraints can be expressed and enforced.

6.2.3 Manual Implementation

In a solution-driven PLE context, manual implementation of assets plays a crucial role. This has both an organizational and a technical facet. The problem space feature model covers the former, the PLiC approach the latter.

Organizational business considerations determine how to implement a new feature's assets. This is where the strength of problem space feature modeling lays. As we indicate in Section 3 and further describe in [18], the decision on how to implement a new feature depends on its attached business values (implementation costs, worth for customer, strategical value, etc.). This technique can both be applied directly to the overall solution feature model as well as be delegated into the feature-models of certain sub–product-lines.

After the organizational decision, if a feature shall be reusable or if not, *technical considerations* come into play. Independently of the decision, the feature may be developed within a PLiC. Each time, it may comprise detailed configuration languages, generation facilities, and the actual implementation. Note, that, although a feature is implemented solution-specific, it will usually still have configuration options for fine-tuning, multiple solution instances, etc. Only its reusability is constrained, as the PLiC makes rigid assumptions about its context, this is the features selected in other sub-domains.

For the actual implementation of variability in a reusable or product-specific way, there exist various possibilities (cf. Table 2). On the one side it is possible to *add*, *change*, and *delete* new artifacts within the reusable asset base. Adding, changing, and deleting is possible on common core assets (C), variation points (VP), and variants (V). On the other side it is possible to *add* new solution-specific variations, to *override* reusable assets, or even to *conceal* common core asset functionality.

	Reusable impl.	Product-specific impl.
Add	Add C, VP, V	Add V
Change	Change C, VP, V	(Override C, VP, V)
Delete	Delete C, VP, V	(Conceal C)

Table 2: Manual Implementation of Assets

Overriding and concealing emerge when there is no suitable variation point in the reusable assets base at a certain location, although one would be needed to perform an adaptation. Usually this is referred to as "patching" and is discouraged; adding an explicit, additional variation point into the reusable asset base and write a variant for it is preferable. This means that, even if a feature shall be implemented product-specific, it still might require adapting the reusable asset base for adding a variation point.

7. DISCUSSION

The PLiC approach is currently in stage of prototypical implementation and evaluation. The interfaces of PLiCs as described above are simple and we have to see if hierarchical composition and black box behavior is sufficient for both configuration and generation. In the following, we discuss the notion of sub-domain, configuration model consistency, scalability, and binding times.

7.1 The Notion of a Sub-domain

The notion of sub-domains is of high relevance in industrial practice of large-scale systems and is used to (hierarchically) structure the overall product line domain. However, there is no generally agreed definition. So, it depends on the context if it refers to problem space, solution space, or organization. In Section 3, we define a sub-domain of a product line as a subpart of the overall product line domain, whereas there is a high cohesion in problem space, solution space, *and* in company-internal organization. Of course, this does not mean there is no interfacing between sub-domains, but that they are relatively stable and clear, in contrast to sub-domain–internal interfacing. This enables us to bundle all artifacts relevant to one sub-domain into a conceptual entity with quite clear interfaces, the PLiC. The problem of partitioning into sub-domains stems from practice and we are convinced that problem space, solution space, and organization have to be considered together to tackle large-scale product line engineering.

7.2 Configuration Model Consistency

From a technical point of view, evaluating the consistency of configuration models is easiest on demand, for example by triggering the evaluation of consistency rules in OCL explicitly. There also exists research about high performance on access/edit constraint checking [9]. The more challenging question is how to manage creation and maintenance of the consistency rules in order to keep the models of heterogeneous types valid. At the moment, we consider using general purpose constraint checking languages, such as OCL, and to assign rule sets to each sub-domain, whereas rules may only access models in their own or in child sub-domains. Checking the consistency of a solution then corresponds to checking the rule set of the root solution-driven product line and all sub-domain rule sets recursively.

7.3 Scalability

The question remains, if such a generic checking infrastructure scales when it comes to large-scale product lines. Next to hard dependencies (requires, excludes), there may be weak ones, for example regarding the influence on execution time or memory usage. These may be covered similar to COVAMOF [19], where weak constraints have attached textual information indicating the impact of a configuration on certain system qualities.

7.4 Binding Times

Having different binding times within the product line could be done via dividing the Check_Configuration service (see Section 6.1) into several services. Each service would represent a binding stage and could introduce stricter constraints. However, we have not elaborated on that issue yet.

8. RELATED WORK

We resemble staged and multi-level configuration of feature models [7] to some extent, as we have a hierarchical model structure that refines from abstract feature to a more and more concrete configuration. However, staged configuration only supports one type of variability model, a feature model. This is the case for many other reported applications, like decision modeling [8], COVAMOF [19], or OVM [15]. In contrast, our approach is free in using various domain-specific modeling languages; we only expect the top level model to be a feature model to link to the problem space.

In contrast to our feature notion, which is rather abstract and driven from problem space considerations, feature-oriented software development (FOSD, [1]) operationalizes a feature as an increment in program functionality that implements a requirement. A sub-domain, as we define it, consists of a set of increments and rules specifying valid combinations (similar to the feature-oriented view on a product line). Ideally, in FOSD, each feature is implemented separately in a so-called feature module to separate the concerns (SoC) of the features. Our approach requires SoC on sub-domain level, while we currently do not explicitly consider SoC within a sub-domain. This means features may be implemented applying strict SoC or not. We aim at implementing SoC on sub-domain-level by classical means, in es-

pecially, providing variation points (e.g., by design patterns) in one sub-domain, while other sub-domains may provide corresponding variants.

Product derivation systems based on version management software [20, 21], manage the reusable asset base (the product line platform) and the reused assets in the actual products separately and preserve dependencies between reusable and reused asset. It follows the assumption that the development of platform and products generally happens independent of each other, but that at some points (e.g., security updates) merges from platform to product assets or vice versa become necessary. In our approach, we try to avoid solution-specific overrides or concealing of assets. Instead, we would prefer to add a variation point to the reusable asset base whenever possible. As, in our application context, the number of concurrent solutions is rather limited (3 to 10) this still seems feasible to us.

9. SUMMARY AND FUTURE WORK

In this paper, we have elaborated and approached the challenges when doing product configuration and generation in a large-scale product line context in solution business. We identified the following four challenges: the need for decentralized configuration, for heterogeneous configuration, for explicit product generation support, and for supporting application engineering as extensive task. We tackle the challenges with the PLiC approach. It resembles the fact that the domain of a large-scale product line is divided into sub-domains. A PLiC (product line component) encapsulates all configuration, generation, and implementation artifacts of a sub-domain. A hierarchical composition of PLiCs then constitutes the overall solution-driven product line. Our approach facilitates decentralized and heterogeneous configuration by allowing multiple models, arbitrary model types, and constraints that may span sub-domains. It gives product generation support by hierarchical delegation of generation calls, and, by integrating a problem space feature model into the derivation process on solution space side, we can give explicit support for application *engineering* as an extensive development process.

Before applying the approach in a real-world context still a lot of research remains to be done. Our next step will be to set up a prototype satisfying the identified characteristics and elaborate on the more technical details of sub-domain–spanning product configuration, consistency checking, and product generation.

Acknowledgments

We thank Christa Schwanninger and Ludger Fiege for their valuable feedback on earlier versions of this paper.

10. REFERENCES

[1] D. Batory. Feature-oriented programming and the AHEAD tool suite. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*, pages 702–703. IEEE Computer Society Press, 2004.

[2] D. Beuche. Variant management with pure::variants. Technical report, pure-systems GmbH, 2006. `http://www.pure-systems.com/fileadmin/downloads/pv-whitepaper-en-04.pdf`, visited 2009-03-26.

[3] J. Bosch. *Design and Use of Software Architectures, Adopting and Evolving a Product Line Approach*. Addison-Wesley, 2000.

[4] J. Bosch. Maturity and evolution in software product lines: Approaches, artefacts and organization. In *Proceedings of the 2nd Software Product Line Conference (SPLC '02)*, pages 257–271, Heidelberg, Germany, 2002. Springer-Verlag.

[5] J. Bosch. Expanding the scope of software product families: Problems and alternative approaches. In C. Hofmeister, I. Crnkovic, and R. Reussner, editors, *Quality of Software Architectures*, Lecture Notes in Computer Science. Springer-Verlag, 2006.

[6] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.

[7] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Staged configuration through specialization and multilevel configuration of feature models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005.

[8] D. Dhungana, R. Rabiser, P. Grünbacher, and T. Neumayer. Integrated tool support for software product line engineering. In *Proceedings of the 22th IEEE International Conference on Automated Software Engineering (ASE '07)*, pages 533–534, New York, NY, USA, 2007. ACM Press.

[9] A. Egyed. Scalable consistency checking between diagrams-the viewintegra approach. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE '03)*, Washington, DC, USA, 2001. IEEE Control Systems Magazine.

[10] Eclipse modeling framework homepage. `http://www.eclipse.org/emf/`.

[11] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, Carnegie Mellon University, Software Engineering Institute, Pittsburgh, PA, Nov. 1990.

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

[13] OpenArchitectureWare homepage. `http://www.openarchitectureware.org/`.

[14] Object Management Group (OMG). Unified modeling language (UML) 2.1.2 superstructure specification. formal/2007-11-02, November 2007.

[15] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, 2005.

[16] K. Schmid. A comprehensive product line scoping approach and its validation. In *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*, New York, NY, USA, 2002. ACM Press.

[17] K. Schmid. *Planning Software Reuse - A Disciplined Scoping Approach for Software Product Lines*. PhD thesis, Stuttgart, 2003.

[18] C. Schwanninger, I. Groher, C. Elsner, and M. Lehofer. Variability modelling throughout the product line lifecycle. In *Proceedings of the ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems, to appear*. Springer-Verlag, 2009.

[19] M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch. COVAMOF: A framework for modeling variability in software product families. In *Proceedings of the 11th Software Product Line Conference (SPLC '07)*, Heidelberg, Germany, 2007. Springer-Verlag.

[20] C. Thao, E. V. Munson, and T. N. Nguyen. Software configuration management for product derivation in software product families. In *Proceedings of the 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, pages 265–274, Washington, DC, USA, 2008. IEEE Control Systems Magazine.

[21] J. van Gurp and C. Prehofer. Version management tools as a basis for integrating product derivation and software product families. In *Proceedings of the Workshop on Variability Management - Working with Variability Mechanisms at SPLC 2006*, pages 48–58. Fraunhofer IESE, 2006.

[22] Eclipse XText homepage. `http://www.eclipse.org/Xtext/`.