

Tool Support for COVAMOF

Marco Sinnema, Onno de Graaf, Jan Bosch

Department of Mathematics and Computing Science, University of Groningen,
PO Box 800, 9700 AV Groningen, The Netherlands,
{m.sinnema | o.de.graaf | j.bosch}@cs.rug.nl, <http://segroup.cs.rug.nl>

Abstract. A key aspect of software variability management in software product families is the explicit representation of the variability. From experience at several industrial software product families we found that tool support for variability modeling techniques requires (1) uniform and first-class representation of variation points and dependencies in all abstraction levels, (2) support intrinsic variability modeling, (3) support for multiple views, (4) complete lifecycle coverage, and (5) support for complex and dynamically analyzable dependencies. Although some existing tools provide support one or two requirements, none supports all five. In earlier work we presented a framework for variability modeling, COVAMOF. The contribution of this paper is a description of Mocca, tool support for COVAMOF, that addresses all five requirements.

1 Introduction

The explicit representation of variability has been identified as a key aspect of software variability management in software product families [4] [6]. In the past few years, several approaches to variability modeling have been developed [1] [2] [3] [6] [11] [18] [20] [22].

In order to fully exploit the benefits of variability modeling techniques, tool support is required to manage the models and product family artifacts [9]. From experience at several industrial software product families [5] [8] [9] we found that tool support for variability modeling in industrial product families requires (1) uniform and first-class representation of variation points and dependencies in all abstraction levels, (2) support for intrinsic variability modeling, (3) support for multiple views on the models, (4) complete software development lifecycle coverage, and (5) support for complex and dynamically analyzable dependencies. Intrinsic modeling of variability means that the product family artifacts themselves capture their own variability model and that each variability realization technique in the artifacts provides the variability modeling elements. The variability modeling elements in intrinsically modeled product families together form the variability model. In the past few years several tools for variability modeling approaches have been developed, e.g. [1], [3] and [18]. Although some tools do address one or two requirements, none supports all five.

In [20], we presented the ConIPF Variability Modeling Framework (COVAMOF), a variability modeling approach that uniformly models the variability in all abstraction layers of a software product family, i.e. in the features, the architecture and the component implementations. It treats variation points and dependencies as first-class citizens and provides means to model the relations between variation points; simple dependencies as well as complex dependencies.

In this paper, we present Mocca, a variability management tool that supports software engineers to manage the variability model of a product family during domain engineering and application engineering. Mocca uses the variability modeling approach from COVAMOF as a basis. The variability model of Mocca is a union of variability information from multiple sources, e.g. intrinsic modeled artifacts and extrinsic models, and can be managed from multiple views. Mocca addresses all the requirements on tool support we presented above. We have validated the applicability of COVAMOF and Mocca at three industrial product families, of which an excerpt is presented in [20].

The structure of this paper is as follows: in the next section we present requirements on tool support for variability management and in section 3 we show to what extent existing tools support these requirements. Section 4 briefly introduces COVAMOF and section 5 presents Mocca, whereas section 6 discusses this tool and concludes the paper.

2 Requirements

The main benefit of product families is that commonalities between products can be exploited, while at the same time the ability to vary the products is preserved. In order to be able to exploit these benefits, tool support is required to manage the variability provided by the product family artifacts [9]. Such tool support should provide means to maintain the models as part of domain engineering and use the models as part of application engineering. From our experience in variability management in software product families [5] [8] [9] and three industrial case studies [9] [20], we found five main requirements on tooling for variability management for industrial product families. In the next section we show to what extent three existing tools support these requirements.

- **R1: Uniform and first-class representation of variation points and dependencies in all abstraction levels:** The uniform and first-class representation of variation points facilitates the assessment of the impact of selections during product derivation and changes during evolution [5]. Industrial experience indicated that most effort during product derivation is on satisfying dependencies [9]. As the first class representation of dependencies can provide a good overview on all dependencies in the software product family, the efficiency of product derivation increases.
- **R2: Support intrinsic models:** [12] reports on the problem of the possible drift between variability models and the product family artifacts. As intrinsically modeled product family artifacts capture their own variability model, changes to these artifacts are in fact changes to the model itself. Tools that support the

intrinsic modeling of the variability therefore address the problem of this possible drift.

- **R3: Support multiple views:** Although there is not yet a general agreement about which views are useful, the reason behind multiple views is always the same [13]: separating aspects into separate views helps people to manage complexity. For medium and large scale software product families, variability models get too complex to manage in one single view. Therefore, software engineers should be able to manage the variability in the product family from different views. Tools should be able to present and allow the user to edit the variability information from these views.
- **R4: Cover complete lifecycle:** Tool support should provide means to manage the product family in all phases of the development lifecycle. For domain engineering, this means the extrinsic and intrinsic models of both pre-runtime and runtime variability should be manageable. For application engineering, this means that for pre-deployment variability the tooling should provide an interface to derive and configure products from the product family and for runtime variability should be configurable and manageable manually by a user or automatically by a software component.
- **R5: Support for complex and dynamically analyzable dependencies:** In industrial product families, there are several types of dependencies. *Simple* dependencies fully specify the restriction on the binding of one or two variation points, e.g. “the binding of variant A1 to variation point A excludes the binding of variant B1 to variation point B”. Dependencies in industrial product families, however, are often more *complex* and typically affect a large number of variation points. Dependencies can furthermore, in many cases, not be stated formally, but have a more informal character, e.g. “these combinations of parameter settings will have a negative effect on the performance of the overall software system”. We refer to dependencies of which the validity can be calculated from the selection of variants of the associated variation points as *statically analyzable* dependencies, e.g. the mutual exclusion of two component implementations. In other cases, when the verification of the validity requires a test of the software system, we refer to the dependencies as *dynamically analyzable* dependencies. Having an explicit overview on the dependencies of all these types improves the product derivation process.

3 Existing tools

Below, we briefly describe three approaches with tool support that model variability in software product families and in Table 1 we show to which extent these approaches address the requirements presented in the previous section, based on the information currently available. From Table 1 we conclude that none of these three approaches fully support the requirements.

- **Building Product Populations with Software Components:** Van Ommering [18] presents how Koala [19] can be used in the context of software product families. Koala is an approach to realize the uniform binding of pre-runtime

and runtime variability of components, independent to its context. Components are recursively specified in terms of first class provided and required interfaces. The variability in the design is specified in terms of the selection of components, parameters on components, and the runtime routing of function calls. The interfaces of Koala components specify whether two components are compatible. There is currently no support for multiple views on the product family variability.

- **A Koala-Based Approach for Modelling and Deploying Configurable Software Product Families:** Asikainen et al. [1] have developed Koalish, an extension to Koala [19] with constructs to specify variability and constraints, and the possibility to select the type and number of components. The constraints can be used to specify logical dependencies between components and interfaces. Tools currently support Koalish for the application engineering stage to configure a Koalish model and generate a configuration to a Koala product model. There is currently no support for multiple views on the product family variability.
- **Mapping Variabilities onto Product Family Assets:** Becker [3] presents an approach in which the representation of variability of the product family is separated into two levels, i.e. the specification and the realization level. The variability on the specification level is defined in terms of variabilities, and on the realization level in terms of variation points. Variabilities specify the required variability and variation points indicate the places in the asset base that implement the required variability. This model contains two types of these variation points, i.e. static (pre-deployment), and dynamic (post-deployment) variation points. Dependencies can be specified in a 1-to-1 manner and are not represented as first-class citizens.

Table 1 The adherence of existing tools to the requirements for tool support for variability management presented in section 2

Approach	R1	R2	R3	R4	R5
Van Ommering [18]	-	+	-	+	-
Asikainen et al. [1]	-	+	-	+	-
Becker [3]	-	-	-	+	-

4 ConIPF Variability Modeling Framework (COVAMOF)

As part of the ConIPF project [7], we developed the ConIPF Variability Modeling Framework (COVAMOF) [20]. In this section, we present a short description of this framework. The aspect of COVAMOF most relevant for this paper is the COVAMOF Variability View (CVV), which is a view on the variability provided by the product family. Below, we first describe the core concepts behind the CVV. Subsequently, we present the structure of the CVV.

4.1 COVAMOF Variability View (CVV)

A product family is divided into three abstraction layers, i.e. features, architecture, and component implementations, and the hierarchy throughout these layers is defined by levels of abstraction. As we briefly mentioned above, the CVV is a view on variability provided by a software product family (See Figure 1). As variation in software product families can occur in all three abstraction layers, the CVV view encompasses the variability on all these layers of abstraction.

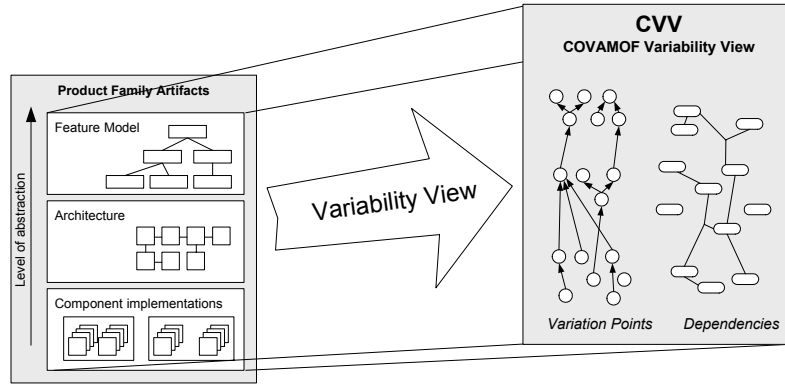


Fig. 1. COVAMOF provides the variability as a view on the product family artifacts in terms of variation points and dependencies

4.2 Structure of the CVV

The main elements the CVV are *variation points* and *dependencies*, which are represented as first class citizens. Below, we briefly describe these elements and refer to [20] for a more elaborate description.

Variation points

Variation points are places in product family artifacts at which there is a choice between multiple options while configuring a product [14], e.g. the choice between two sub-features, the choice of including an optional component in the architecture, and the configurable value of a startup parameter. Variation points in the CVV are a view on these variation points. Each variation point in the CVV is associated with an artifact in the product family, e.g. an architectural component. There are five types of variation points in the CVV, i.e. optional, alternative, optional variant, variant, and value.

- **Optional:** An *optional* variation point is the choice of selecting zero or one from the one or more associated variants.
- **Alternative:** An *alternative* variation point is the choice between one of the one or more associated variants.

- **Optional variant:** An *optional variant* variation point is the selection (zero or more) from the one or more associated variants.
- **Variant:** A *variant* variation point is the selection (one or more) from the one or more associated variants.
- **Value:** A *value* variation point is a value the can be chosen in a predefined range.

The variation points in the CVV specify, for each variant or value, the actions that should be taken in order to realize the choice, for that variant or value, in the product family artifacts, e.g. the selection of a feature in the feature tree, the adaptation of a configuration file, or the specification of a compilation parameter. These actions can be specified formally, e.g. to allow for automatic component configuration by a tool, or in natural language, e.g. a guideline for manual steps that should be taken by the software engineers.

Variation points that have no associated realization mechanism in the product family artifacts are realized by variation points on a lower level of abstraction, e.g. an optional architectural component that realizes the choice between two features in the feature tree. These realizations are represented by *realization relations* between variation points in the CVV. This relation provides traceability and a hierarchical structure on the variation points in the product family.

Dependencies

Dependencies in the context of variability are restrictions on the variant selection of one or more variation points, and are indicated as a primary concern in software product families [15]. These dependencies originate, amongst others, from the application domain (e.g. customer requirements), target platform, implementation details, or restrictions on quality attributes. These dependencies are modeled in the CVV and are associated to one or more variation points in the CVV and can be formulated formally, e.g. the mutual exclusion of two features, or informally, e.g. the estimated memory usage based on the variant selection of five parameter values.

Dependencies furthermore are not isolated entities. The process of resolving one dependency may affect the validity of other dependencies. We refer to this as dependency interaction. *Dependency interactions* in the CVV specify how two or more dependencies mutually interact. The interaction provides a description of the origin of the interaction and specifies how to cope with the interaction during product derivation.

Summary

The structure of main entities the CVV we described above is presented in Figure 2.

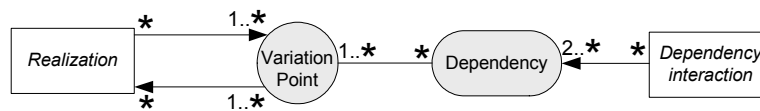


Fig. 2. The metamodel of the main entities in the CVV.

5 Mocca: Tool Support for COVAMOF

As none of the approaches presented in section 3 addresses all requirements in section 2, we have designed and implemented a new tool, called Mocca. The Mocca tool suite is designed to manage the COVAMOF Variability View (CVV) in all phases of the development lifecycle. From the user perspective, the functionality of Mocca boils down to the development and evolution of the CVV during domain engineering and the usage of the CVV during application engineering. It furthermore supports automatic configuration at both pre-compile time and runtime. In this section, we present the design and implementation of Mocca.

5.1 Design

The basic idea behind Mocca is presented by the architecture in Figure 3. The architecture consists of the *CVV Platform*, zero or more *Mechanism* plug-ins and zero or more *Controller* plug-ins. In section 2 we presented five requirements on tool support for variability management. In order to support requirement R1 and R5, Mocca uses the CVV as representation of the variability model. In the CVV, dependencies and variation points are therefore represented as first-class citizens uniformly on all abstraction levels [20].

In order to address requirement R2, the CVV Platform maintains the CVV and provides an interface to manage the model. It retrieves model information from Mechanism plug-ins and presents this information as one model to the Controllers, as explained in Figure 4. As different Mechanism plug-ins can provide variability information from different sources, both intrinsic modeled artifacts and extrinsic models are supported.

The CVV Platform allows for multiple Controller plug-ins, in order to address requirement R3. Figure 4 shows that these Controller plug-ins allow for viewing and editing the CVV from different views, e.g. the variation point view or dependency view.

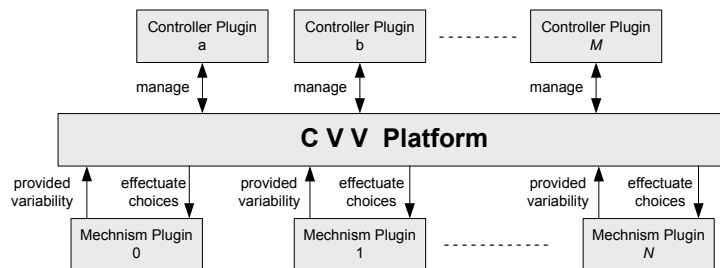


Fig. 3. Simplified view of the Mocca architecture

Related to requirement R3, Mocca functionality can both be provided statically, for the development and pre-deployment configuration of the product family, and dynamically, for runtime configuration and maintenance of the product family variability. Therefore, the CVV platform should not only be realized as separate tool,

but also as part of a running product. Note that the implementation of the plug-ins used during static and dynamic application may vary to serve the different types of usage.

As presented in Figure 3, the Mocca architecture consists of three main entities:

- **Mechanism Plug-in:** A Mechanism plug-in is associated to one or more variability realization mechanisms in the product family. It exports the provided variability related to the associated mechanism. This information can on the one hand come from an extrinsic model, e.g. represented in XML, which contains the provided variability. On the other hand, in order to support requirement R2, this plug-in can read the provided variability from intrinsically modeled artifacts. The plug-in can furthermore effectuate choices made within the CVV Platform for variation points that are associated to the Mechanism plug-in. The implementation of a Mechanism depends on the variability realization mechanism and the associated binding time.
- **CVV Platform:** The CVV Platform maintains the CVV and provides an interface to manage the model. It retrieves model information from the associated Mechanisms and presents this information as one model to the Controllers (see also Figure 4). It furthermore notifies the Controllers and Mechanisms when the model changes, e.g. due to modifications during evolution or selections during product derivation.
- **Controller Plug-in:** In order to support requirement R3, multiple Controller plug-ins manage the CVV Platform from different points of view. A Controller can be a user interface to a software engineer (for manual development and configuration), an interface to an external software system (for automatic configuration), or an independent component that manages the CVV, e.g. based on runtime measurements of a running software system. Modifications made by the user to the model are passed on to the CVV Platform.

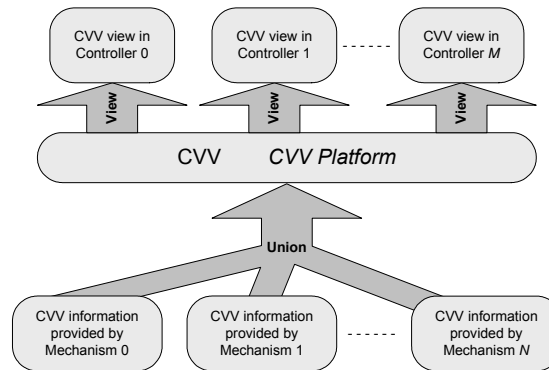


Fig. 4. The CVV Platform presents the CVV information provided by the Mechanism plug-ins as one model to the Controllers. The Controllers manage the CVV in the CVV Platform from one viewpoint.

5.2 Implementation

Up to now, we have realized a subset of the Mocca design, i.e. the CVV Platform, two Controller plug-ins and two Mechanism plug-ins (See also Figure 5). Currently, the Mocca implementation supports pre-runtime development en configuration of the product family. We have implemented all functionality in the Java [16] programming language, partly as stand-alone components, partly as extensions to the Eclipse Platform [10]. Eclipse allows for a tight coupling between the source code and Eclipse plug-ins. As we have implemented Mocca as an Eclipse plug-in, Mocca provides a tight coupling between the source code and the variability model of a product family.

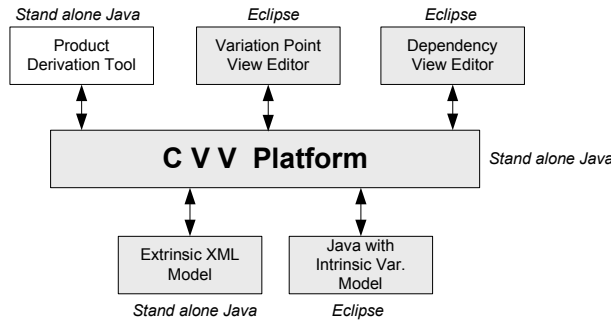


Fig. 5. Mocca Deployment Architecture. In addition to the CVV Platform, we have realized two Mechanism and two Controller plug-ins. We are currently working on the third Controller plug-in, i.e. the Product Derivation Tool.

A more detailed description of the components in the Mocca deployment architecture, shown in figure 5, follows below.

- **CVV Platform:** The CVV Platform is implemented as a stand alone Java library that can be instantiated in a tool for product derivation and evolution of a product family.
- **Variation Point View Controller:** In this plug-in, the CVV can be managed graphically in the Variation Point View. The Variation Point View in the CVV provides the software engineers with an overview on the variation in all abstraction levels of a product family in terms of variation points. The realization and artifact relations provide the structure on the set of variation points. In this view, dependencies are attributes of the variation points. This plug-in has been implemented in Java as an extension to the Eclipse Platform. Figure 6 shows a screenshot of this plug-in as used during domain engineering.
- **Dependency View Controller:** In this plug-in, the CVV can be managed graphically in the Dependency View. The main entities in the Dependency View are the dependencies and the dependency interactions that provide the structure on the set of dependencies. Variation points in this view are attributes of the dependencies. The Dependency View provides software engineers with an overview on the most critical dependencies, e.g. based on their type, number of associated variation points or number of dependency interactions, which can

be used to develop a strategy to resolve the dependencies during product derivation. This plug-in has been implemented in Java as an extension to the Eclipse Platform. Figure 6 shows a screenshot of this plug-in as used during domain engineering.

- **Extrinsic XML Model Mechanism:** This Mechanism plug-in reads and writes extrinsic CVV information, exports it to the CVV Platform, and maintains the link between the information in the model and the real product family artifacts. If, during product derivation, a variant is selected of a variation point associated to the Mechanism plug-in, the choice is effectuated in the real product family artifacts based on the information in the CVV variant.
- **Java with Intrinsic Variability Model Mechanism:** This mechanism plug-in reads the CVV information from java [16] source files and exports this information to the CVV Platform.

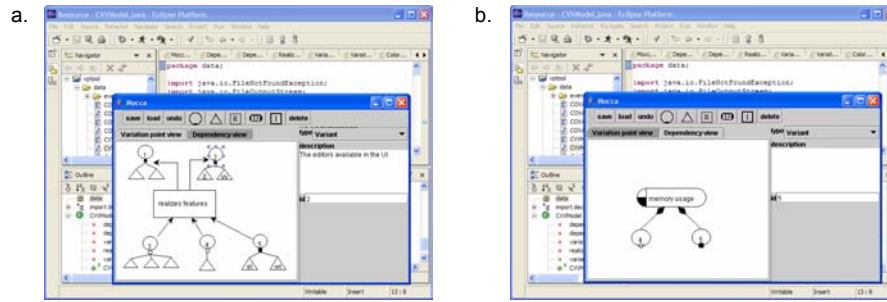


Fig. 6. Screenshots of the Variation Point View Controller (a.) and the Dependency View Controller (b.) as Eclipse plug-in

6 Conclusion and future work

In this paper, we have introduced our approach to tool support for variability management in software product families. We presented five requirements on tool support for variability management and showed that none of the existing tools supports all of these requirements. We briefly introduced the ConIPF Variability Modeling Framework (COVAMOF) [20], and our approach to tool support for this framework, which addresses all five requirements:

- **R1:** As Mocca uses the CVV as underlying variability model, dependencies and variation points are represented as first-class citizens uniformly on all abstraction levels [20]. Therefore Mocca fully supports requirement R1.
- **R2:** In order to address R2, Mocca consists of a platform, the CVV Platform (see also Figure 4), which maintains information provided by several Mechanisms that provide parts of the model information. These mechanisms are plug-ins in the CVV Platform and are directly associated to the modeled artifacts. These artifacts can be modeled intrinsically or extrinsically.

- **R3:** In order to address R3, the CVV Platform provides an interface to Controller plug-ins, which can manipulate the CVV from one view, e.g. the Variation Point View and the Dependency View.
- **R4:** Information on the variability is maintained in the CVV, as pre-runtime stand alone tool and as runtime library next to the running product. The design of Mocca therefore also addresses R4.
- **R5:** As Mocca uses the CVV as underlying variability model, complex dependencies and dynamically analyzable dependencies are supported. Therefore Mocca fully supports requirement R5.

As part of the ConIPF project [7], we have validated the applicability of COVAMOF and Mocca at three industrial product families, of which an excerpt is presented in [20]. We are currently extending the implementation of Mocca with tool support for all functionality necessary to allow for pre-runtime automatic configuration and runtime (re-)configuration and maintenance of a product.

Acknowledgements. This research has been sponsored by ConIPF [7] (Configuration in Industrial Product Families), under contract no. IST-2001-34438. The ConIPF project aims to define and validate methodologies for product derivation that are practicable in industrial applications.

7 References

1. T. Asikainen, T. Soininen, T. Männistö: A Koala-Based Approach for Modelling and Deploying Configurable Software Product Families, 5th Workshop on Product Family Engineering (PFE-5), November 2003, to be published in Springer Verlag Lecture Notes on Computer Science no. 3014, 2004.
2. F. Bachman, M. Goedicke, J. Leite, R. Nord, K. Pohl, B. Ramesh, A. Vilbig: Managing Variability in Product Family Development, 5th Workshop on Product Family Engineering (PFE-5), November 2003, to be published in Springer Verlag Lecture Notes on Computer Science no. 3014, 2004.
3. M. Becker: Mapping Variability's onto Product Family Assets, Proceedings of the International Colloquium of the Sonderforschungsbereich 501, University of Kaiserslautern, Germany, March 2003.
4. J. Bosch: Design & Use of Software Architectures, Adopting and Evolving a product-line approach, Addison-Wesley, ISBN 0-201-67494-7, 2000.
5. J. Bosch, G. Florijn, D. Greefhorst, J. Kuusela, H. Obbink, K. Pohl: Variability Issues in Software Product Lines, Proceedings of the Fourth International Workshop on Product Family Engineering (PFE-4), pp. 11–19, 2001.
6. M. Clauss: Modeling variability with UML, GCSE 2001 - Young Researchers Workshop, September 2001.
7. The ConIPF project (Configuration of Industrial Product Families), <http://segroup.rug.nl/conipf>.
8. S. Deelstra, M. Sinnema, J. Bosch, A Product Derivation Framework for Software Product Families, Proceedings of the 5th Workshop on Product Family Engineering (PFE-5), Springer Verlag Lecture Notes on Computer Science Vol. 3014 (LNCS 3014), pp. 473-484, May 2004.
9. S. Deelstra, M. Sinnema, J. Bosch: Product Derivation in Software Product Families; A Case Study, accepted for the Journal of Systems and Software, 2003.

10. Eclipse Platform, <http://www.eclipse.org>.
11. H. Gomaa, M.E. Shin: Multiple-View Meta-Modeling of Software Product Lines, 8th International Conference on Engineering of Complex Computer Systems (ICECCS 2002), IEEE Computer Society 2002, ISBN 0-7695-1757-9, pp. 238-246, 2002.
12. J. van Gurp, J. Bosch: Design Erosion: Problems & Causes, *Journal of Systems and Software*, 61(2), pp. 105-119, March 2002.
13. C. Hofmeister, R. Nord, D. Soni: *Applied Software Architecture*, Addison-Wesley, 2000.
14. I. Jacobson, M. Griss, P. Jonsson: *Software Reuse. Architecture, Process and Organization for Business Success*. Addison-Wesley, ISBN: 0-201-92476-5, 1997.
15. M. Jaring and J. Bosch: Variability Dependencies in Product Family Engineering, accepted for the 5th Workshop on Product Family Engineering (PFE-5), November 2003, to be published in Springer Verlag Lecture Notes on Computer Science no. 3014, 2004.
16. Java Technology, <http://java.sun.com>.
17. M. Jazayeri, A. Ran, F. van der Linden: *Software Architecture for Product Families: Principles and Practice*, Addison-Wesley, 2000.
18. R. van Ommering: Building Product Populations with Software Components, in *Proceedings of the 24th International Conference on Software Engineering (ICSE'02)*, May 2002.
19. R. van Ommering, F. van der Linden, J. Kramer, J. Magee: The Koala Component Model for Consumer Electronics Software, *IEEE Computer*, pp. 78-85, March 2000.
20. M. Sinnema, S. Deelstra, J. Nijhuis, J. Bosch, COVAMOF: A Framework for Modeling Variability in Software Product Families, accepted for the Third Software Product Line Conference (SPLC 2004), August 2004.
21. M. Svahnberg, J. Gurp, J. Bosch: A Taxonomy of Variability Realization Techniques, technical paper ISSN: 1103-1581, Blekinge Institute of Technology, Sweden, 2002.
22. S. Thiel, A. Hein: Systematic integration of Variability into Product Line Architecture Design, *Proceedings of the 2nd International Conference on Software Product Lines (SPLC-2)*, August 2002.