

Combining Software Product Lines and Structure-based Configuration – Methods and Experiences

Lothar Hotz¹, Thorsten Krebs², and Katharina Wolter²

¹ HITeC c/o Fachbereich Informatik, Universität Hamburg
Hamburg, Germany

`hotz@informatik.uni-hamburg.de`

² LKI, Fachbereich Informatik, Universität Hamburg
Hamburg, Germany

`{kwolter|krebs}@informatik.uni-hamburg.de`

Abstract. In this paper, we discuss the use of structure-based configuration methods for deriving products in the context of software product lines. Modeling techniques for features, artifacts as well as procedural knowledge and inference methods are presented and illustrated with the configuration tool KONWERK. The presented material is a result of our practical work with product lines at our industrial partners.³

1 Introduction

Software development in general is a *design task* where new artifacts (software components, programs, exe-files, libraries etc.) are developed. The result of each development process is one software product. Using software product lines, software development is divided into two processes: domain engineering and application engineering. In domain engineering reusable artifacts are developed for the concerned product family, i.e. not for only one product but for a set of products. During application engineering products are constructed by selecting a set of these artifacts. Thus, an important part of application engineering is a *composition task* - assembling a set of artifacts that fit together and that as a set meets the customer requirements. This task is still very complex since the set of points where variability occur can be very large and several dependencies can exist between them. Furthermore, there is a lack of support for application engineering (see e.g. [7]). As a result, for instance, functionality is implemented anew where reuse would have been possible or impacts of decisions are not known or overlooked because the large set of components is hardly manageable.

Structure-based configuration from the field of AI is a method that supports the composition of products out of a given set of components. In software product lines such a set of components is developed in domain engineering. Based

³ This research has been supported by the EU under the grant IST-2001-34438, ConIPF - Configuration in Industrial Product Families.

on the separation of domain engineering and application engineering structure-based configuration can be used to support the latter. In this paper, we describe different support opportunities and services for application engineering enabled by structure-based configuration.

The remainder of this paper is organized as follows: first we present requirements and challenges which arise when software components of a product family are composed in combination with an example, second we give a brief introduction to structure-based configuration (Section 3). Next, we describe the different support opportunities and services enabled by structure-based configuration and illustrate them with the tool KONWERK (Section 4). In Section 5 the integration of the approach described in this paper with tools already used in software development is outlined. Finally, we give a discussion and some experience we made.

2 Challenges in application engineering

The work presented in this paper is part of the ConIPF project. In the first phase of this project case studies have been performed at the two participating industrial partners. In the following, we describe some of the problems and challenges these companies are faced with during application engineering. In Section 4 we will explain how structure-based configuration can be used to offer support in these areas.

Manage variability. Because of the lack of methodological support for application engineering approaches like "copy and modify" are used. Following such manual processes the solutions can easily be *inconsistent*, e.g. the selected components do not fit because their interfaces do not match. Moreover, the solution can be *incomplete*, i.e. necessary components are missing in the product. Finally solutions can be *incorrect*, i.e. the components included in the product do not realize the needed functionality.

Process. The task of application engineering is very complex because of the large number of decisions on different levels of abstraction involved in the process. Furthermore, these decisions depend on each other. This can make the process slow and error-prone even if no new development is involved.

Dependencies. The selection or change of a specific component, for instance, often has effects on other parts of the system. During application engineering it is very important to have information about all these dependencies. In many cases however this information is only known by some experts.

Evolution. System components evolve over time and new features and components might be integrated in the product line. Handling evolution is a big challenge, for instance, the same component in different versions may require the existence of different other components.

For illustrating our approach throughout this paper, we use an example which comes from the domain of *Car Periphery Supervision (CPS) systems* as introduced by [18]. A CPS system consists of automotive systems that are based on

sensors installed around the car to monitor its local environment. The recording and evaluation of sensor data enables different kinds of high-level applications, which can be grouped into safety-related and comfort-related applications. One example of such applications is **Pre-Crash Detection**. Based on sensor information with Pre-Crash Detection it is possible to estimate the time, area, and direction of an impact before the crash happens. This enables e.g. adjusting trigger points of specific airbags in different locations in the car appropriately for the estimated crash situation. Further examples are Blind Spot Detection and Adaptive Cruise Control.

3 Structure-based configuration

In this section we shortly describe our approach to structure-based configuration (SBC). Similar approaches are presented in [17, 16, 12].

SBC supports the composition of complex products out of a given set of components. The knowledge needed for this task is defined in a *configuration model*. Basic modeling facilities enable the differentiation between three knowledge types (compare [4]):

Conceptual knowledge includes *concepts* to model domain objects with *parameters*, and *taxonomic* and *compositional relations* between these objects, as well as restrictions between arbitrary concepts and their dependencies (by means of *constraints*).

Procedural knowledge declaratively describes the *configuration process* – i.e. the order in which configuration decisions are processed.

A task specification describes the desired product capabilities and already known components (e.g. 3rd-party components).

Parameters are specified by *value descriptors* of type string, integer, float, sets of strings, integers and floats, and intervals of integers as well as floats. Complex dependencies can be defined based on the compositional (e.g. *has-parts*) and taxonomic (i.e. *is-a*) relation and constraints between arbitrary concepts and concept properties. The taxonomic relation is a *strong* taxonomy, i.e. given a superconcept *o* and a subconcept *u* all property values (parameters or relations) which are defined in *u* must be subsets of the related property values of *o*. If there is no related property value in *o* (e.g. because the property is not specified for *o*) the property value of *u* is per default a subset. The configuration model can be seen as an implicit enumeration of admissible configuration solutions in a given domain.

To cope with this complexity, a well-structured configuration procedure – as explained in the following – is needed. For each modeling facility inference methods are provided that support the automatic derivation of configurations by using the configuration model. The configuration process is performed incrementally. Each step represents a configuration decision and the calculation of its logical impacts. Thus, in an interactive configuration process some decisions are made by the user and some are inferred by the configuration system. A decision

is either to set a parameter (*parameterization*), to decide which parts an aggregate has (*decomposition*), to decide what aggregate a part has (*integration*), or to decide if an instance belongs to a more specific concept of the taxonomy (*specialization*). The first decision in a configuration process is defining the configuration goal – i.e. selecting a concept definition as task specification. By inferring impacts of configuration decisions, a partial configuration (consisting of concept instances) is computed.

After each step, optionally *global mechanisms* can be applied for computing new values for specific properties (p_g), testing, simulating or checking consistency of the product – e.g. with constraint techniques (see also [8]). *Global* means that the entire partial configuration is examined instead of only examining one concept instance the configuration decision is directly concerned with. Through constraints for example it is possible that other concept instances are affected by a configuration decision. If the global mechanism cannot find a valid value for a property, a conflict is raised and backtracking in the configuration process is performed. After that other values for certain properties can be chosen and the global mechanism can be applied again. Thus, the p_g properties are computed again, they are *iteratively determined*.

For getting consistent configurations following *consistency rule* for decision making is specified in structure-based configuration like it is provided here: *Property values can only be restricted not enhanced*. This means first, for a decision possible values, which can be chosen, are taken from the concept definition. For example, if in a parameter p of a concept the value descriptor is specified to be the interval $[0\ 10]$ only values in this range can be chosen for the decision! Thus, the model determines possible choices. Furthermore, if a value is selected, e.g. to be 5, for parameter p in a configuration step, it is not possible to change this value in a later configuration step *directly* e.g. to 6. It is only possible to go back in the configuration process (i.e. to perform *backtracking*) to the point where the previous decision was made for parameter p , namely just before 5 is selected, in this configuration step the current value was the value from the model, namely $[0\ 10]$. Here 6 can be chosen, because 6 is in that interval. Various backtracking mechanisms are provided in [5]. Because of this consistency rule a logical based process is performed and only monotonic decisions can be made. Furthermore, the configuration process is deterministic, i.e. if same configuration steps are executed in a different order, the resulting configuration is the same [15].

The result of a configuration process is a complete configuration: a *description* of the generated solution. A complete configuration consists of concept instances that are completely specified. A concept instance is *completely* specified when there is no subconcept this instance can be specialized to and there are no properties (i.e. parameters and compositional relations) that can be further specified. As discussed before changing of specified property values can only be done by backtracking.

In our agenda-based approach, all configuration decisions needed for getting a complete configuration are collected in an agenda. For each concept instance that is part of the partial configuration and that is *incompletely* specified, for

each decision related to that instance a corresponding agenda entry is created. Which decisions are put on the agenda, the order in which configuration decisions can be made, and how the value for a decision is determined can be defined in the procedural knowledge.

We use the configuration knowledge modeling language (CKML) for paper presentations. CKML includes all the modeling facilities described above and is implemented as the language BHIBS in the configuration tool KONWERK [6]. An example of a CKML notation is given in Figure 1. This defines the feature *Services-during-Operation* to consist of 2 to 5 features which have exactly one *Activation-services* and one *Supervision* and can have as optional *Monitoring-Services*, *Power-Management-Service*, and a *User-Interface*.

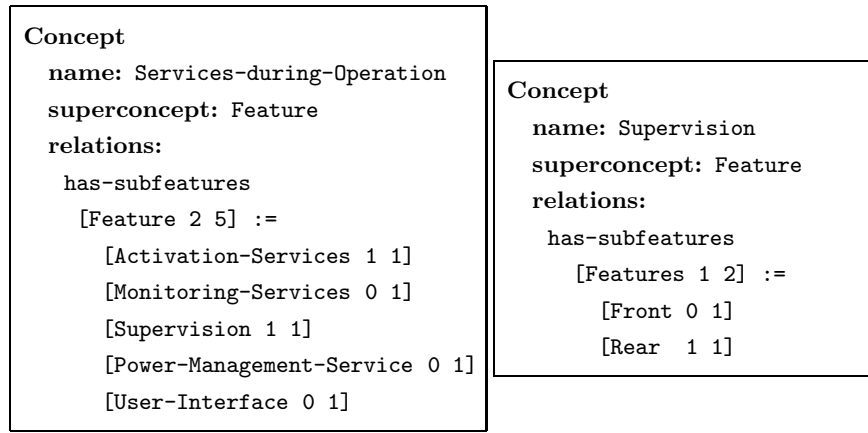


Fig. 1. Example of two application-specific concepts in CKML

4 Support Facilities

In this section we describe what services can be provided for application engineering when methods of structure-based configuration are used. First we consider how features and artifacts can be modeled by the means given in Section 3. Then, we demonstrate with an example how configuration models are used for product derivation (Section 4.2). How a declarative modeling of the product derivation process can be done is explained in Section 4.3.

4.1 Modeling of Features and Artifacts

Features and artifacts are represented as concepts of CKML (see Figure 2). Relations between features and artifacts are mainly the specialization relation or compositional relations like *has-subfeatures* or *has-parts*. A further important relation is the *is-implemented-by* relation, which is used to define the mapping between features and artifacts and the *requires* relation, which is used between features or artifacts respectively. Also the *is-implemented-by* and the *requires*

relation can be expressed by compositional relations between aggregates and parts. Those compositional relations ensure in the context of configuration, that if a description of the aggregate is generated for the configuration result also a description of the parts are generated. This is also the meaning of the *is-implemented-by* and the *requires* relation, i.e. if a description of e.g. *Parking Assistance* is in the final result, also descriptions of *Distance Measurement* and *Rear Sensor* are required. Thus, compositional relations can also be used for these relations. Because compositional relations have number restrictions also exclude relations can be modeled.

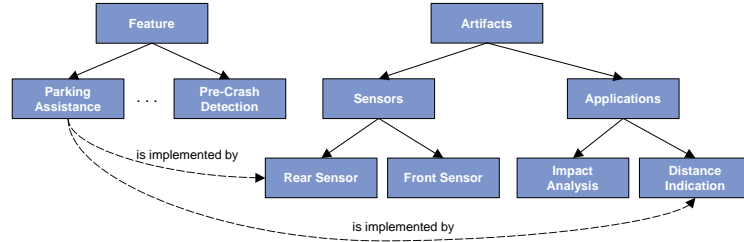


Fig. 2. Features and Artifacts modeled in the configuration model.

4.2 Supporting Automated Derivation

For illustrating how the derivation is supported by automatic inference methods we consider following complex integration step: Given the customer chooses parking assistance and further specifies that he does not only want backward but also forward assistance, an instance of the *Front* feature is automatically created. Since this *Front* belongs to *Supervision* (see Figure 1), an concept instance of *Supervision* is also generated. A supervision is mandatory for *Services-During-Operation* and thus is automatically integrated there. Given another requires relation, which specifies, that when selecting the parking assistance feature the existence of a user interface is required, an instance of the feature *User-Interface* is created. However, since the *User-Interface* is an optional part of the services during operation, it is not automatically integrated. This situation is given in Figure 3. Here the existing supervision concept instance *Supervision-2* is already integrated, the existing concept instance of user interface (*User-Interface-2*) can be selected to be integrated, the *Activation-Services* are already selected to be exact 1 because the model determines this value (see Figure 1), and the other services still have to be configured.

This example also shows the use and processing of multiple compositional relations. Here the *User-Interface* has two relations: one is *Services-During-Operation has-subfeature User-Interface* and the other is *Parking Assistance requires User-Interface*. Assuming the latter was processed first, when configuring the *has-subfeature* relation of *Services-During-Operation* not a new instance of *User-Interface* should automatically be generated but the existing *User-Interface* should be usable and can indeed be used. Thus, *User-Interface* is shared by two

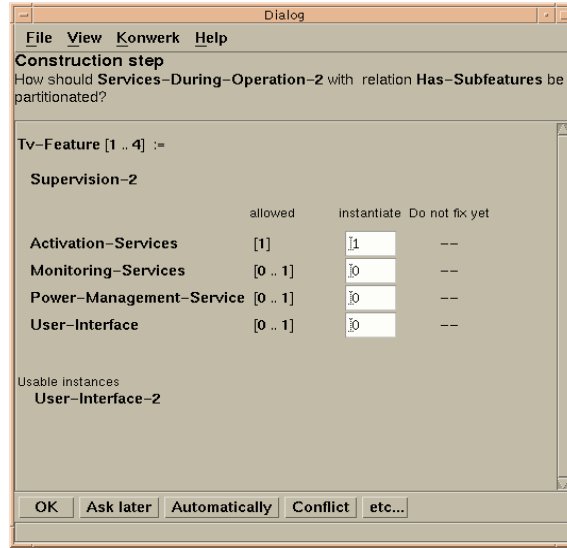


Fig. 3. Integrateable Feature Instance (Snapshot of KONWERK)

aggregates: one of type *Services-During-Operation* and another of type *Parking-Assistance*.

4.3 Declarative Modeling of the Process

Based on these automatic inferences the composition of a product that is complete, correct and consistent can be supported. All admissible products are implicit defined in the configuration model. The product derivation process is performed incrementally. Each step consists of a decision and the calculation of its logical impacts, i.e. in each step a choice is made for one variation point and the effects of this choice are computed. Thus, because of this incremental approach it is possible to provide feedback to the user after each decision. If, for instance, the application engineer selects the feature *parking assistance* (see Figure 2), then the configuration system infers that the components *rear sensor* and *distance indication* are needed because of the is-implemented-by relation in the configuration model. Instances of the corresponding concepts are produced and this effect of the user's decision can be displayed to support the application engineer.

By comparing the solution derived so far with the configuration model it is possible to determine decisions still necessary to complete the product derivation. That is, it is possible to compute a list of variation points for which choices have to be made. These decisions are collected in an agenda and can be presented to the application engineer to visualize the task. Furthermore, it can be ensured that decision cannot be overlooked or forgotten. Additionally, it is possible to structure the product derivation process, e.g. according to the different levels of abstraction. The decisions are sorted according to these levels of abstraction and the application engineer can switch between different agendas. In

the CPS domain, for example, one agenda with decision concerning features and two other agendas with decisions concerning hardware and software components respectively can be used.

Furthermore, it is possible to ensure that restrictions concerning the order of decisions cannot be ignored. If, for example, a set of decisions must be made in order to make certain other decisions, this can be defined in the configuration model. During product derivation the configuration ensures that the defined order is respected. In the CPS domain, for instance, the product needs to be calibrated, i.e. a set of parameter values that make the software function optimally in its run-time environment is iteratively determined. This step cannot be executed unless all other decisions about the product have been made.

If the application engineer starts with making decisions on a high level of abstraction (e.g. about product features), several decisions on a lower level of abstraction (e.g. about components) can already be inferred by the configuration system based on the knowledge defined in the configuration model. Thus, only some of the decisions are made by the application engineer and others are derived by the system leading to a more efficient derivation process.

5 Combination with Other Software Development Tools

In this section, a principle approach for integrating tools and languages known in structure-based configuration with other tools and languages probably existing in a software development and reuse environment are presented. Because there exists a diversity of such tools (like Eclipse, Rational Rose, DOORS, MKS, further on called SE-tools) we give a general or principal view in this section which is based on mappings between representations.

5.1 Modeling

SE-tools are of major interest for *automatic* transformation of data, if already *structured* models are represented with them. If not, only non-automatic mappings to domain models can be processed. However, representatives of the knowledge types which are represented with SE-Tools are first mapped to CKML. If knowledge that can be mapped is identified, a mapping can be defined from the representation of the SE-Tool (typically some XML notation) to a representation of a configuration tool (typically some other XML notation), e.g. by using some kind of XSL transformation. Other knowledge has to be modeled manually with the configuration tool.

Furthermore, software-engineering models that are already strongly structured like feature models, enable automatic transformations of those into a configuration model. This is illustrated with the following example. Given a feature model represented e.g. with DOORS this model maps directly to aggregate relations of a configuration model represented with CKML. However, in DOORS no parameters can be represented, thus, typically relations are used for representing parameters. For maintainability, adequacy and a component-oriented modeling a

Concept	
name:	Steering-assistance
superconcept:	Feature
parameters:	
	Number-of-Moves {Three Two}

Fig. 4. Representing Features with parameters

representation in a configuration model would use parameters. In Figure 4 a parameter representation is shown, where in DOORS additional subfeature would be included.

5.2 Application Engineering

The result of a traditional configuration process is an abstract description of the created solution while in application engineering the solution itself, i.e. the software product is produced. Thus, to support application engineering two processes need to be integrated: the configuration process and the realization process (e.g. code generation, compilation or calibration). For example, code cannot be generated or compiled unless it has been selected in the configuration process. Since these two processes are dependent on each other they need to be synchronized. This can be achieved by means of the procedural knowledge defined in the configuration model. It is possible to initiate the execution of external mechanisms e.g. for compilation. The synchronization of the two processes can be achieved by defining preconditions for the execution of these external mechanisms.

Configuration management (CM) is already used in software development to manage a set of changing artifacts. Using product lines one has not only several versions in time but also different variants to handle. Current CM tools are not sufficient for handling this latter aspect as Muthig states in [13]. On the other hand, configuration tools from the field of AI are not laid out for version management, but can handle versions and variants in the taxonomical hierarchy. However, for generating applications these two technologies must be further integrated.

6 Discussion and Experiences

Besides the challenges in application engineering presented in Section 2, in this section we discuss further experiences made during the ConIPF project.

Modeling and configuring domain objects and considering the application engineering processes at our project partners Bosch and Thales we found that it is *in fact possible to model and configure features and artifacts of a product line with the structure-based configuration methodology*. We currently verify this by applying this methodology in experiments. For this the structure-based configuration tool *EngCon* [1, 14] is used. At Bosch the CPS domain including features

and artifact is modeled. Thus, by selecting certain features, the tool automatically selects the appropriate artifacts. This is also realized by taking constraints, like "if one of four presenting zones of a human interface is configured to be of a certain type, than use the same type for the other zones" into account. The experiments show the principle applicability of the structure-based configuration approach to product derivation. There are a limited number of products in the CPS experiment but with a large number of variety. By using structure-based configuration tools it is now possible to automatically derive products in the CPS domain. However, currently we extend the size of the experiments for getting industrial-realistic scenarios. Earlier work in [14] show that thousands of concepts can be handled by such configuration systems and be used for configuration.

One characteristic of configuring in product lines is that the *number of general types is high*, like context descriptions, features, software components, hardware, architecture, views, calibration parameters. Because the structure-based configuration approach is domain-independent, such distinct types can easily be modeled and configured.

A further characteristic of those domains is that *several compositional relations* are necessary between domain objects, not only one *has-parts* relation like it is often the case in hardware domains. For example, a *has-feature*, *has-subfeature*, *require* relations between different types of features or artifacts have to be used. However, by looking at the configuration process, those relations have similar semantics — namely creating a feature or artifact description when a certain feature or artifact is already in the partial configuration. For example, the *a requires b* relation ensures that if *a* is in the resulting description then also *b* should be there. This is similar for *has-subfeatures*, if *a has-subfeatures b* then if *a* is in the resulting description also *b* is in. The opposite meaning is given by the *exclude* relation. If *a excludes b* then if *a* is in the resulting description than *b* should not. This is typically modeled in structure-based configuration by using number restrictions with the value [0 0] in an aggregate description.

Furthermore, *sharable parts* are necessary, e.g. for modeling libraries, which are used by several software applications. Also features can be qualities of several features or artifacts, like the component which realizes the feature, an aggregate which entails the component and the product as a whole [11]. Thus, a feature can be shared by several other features or artifacts. However, the sharing is automatically managed by a structure-based configuration system.

At our industrial partners only a *small number* of parameters are used for modeling features and software components. Such domain objects are mainly described by their types and compositional relations. Exceptions are only calibration parameters for specific software modules. This is a further difference to hardware domains.

The facility for declaratively describing the configuration process is especially necessary for *representing distinct activities and phases* of the product derivation process (like context configuration, feature configuration etc.). The inevitable aspect of evolution in software product lines can be supported by analyzing the

configuration model and by systematically considering evolution operations on the model. This aspect is worked out in [9].

Also the necessity to *learn the way of modeling* is an experience we made. For example, the modeling is only done on a concept level, i.e. instances are not modeled directly, they are created automatically by the configuration tool. This is necessary for abstracting from a large number of instances, but is sometimes difficult for those not familiar with these terms. Thus, we apply several tutorials and hot-line work for supporting the industrial partners in their modeling task.

7 Related Work

[12, 2] also use methods of structure-based configuration for product derivation in software product lines. However, our approach includes procedural knowledge for defining the configuration process a-priori and is tested by the existing tools KONWERK [6] and EngCon [1].

[18, 7] have suggested a feature-based approach for deriving products in software product families. This approach is similar to ours where we additionally map this approach to structure-based configuration and use those tools.

Other methods that have been built for enhancing reuse strategies in product lines are e.g. defined in [10, 3]. These deal with building up a reusable asset store and feature models that ease the selection for functionality to be included in the product derivation process. Such methods can be supported with our structure-based approach presented in this paper. Therefore, the configuration process absorbs the task of selecting and combining reusable assets.

8 Summary

There exist several methods known in structure-based configuration, like declarative modeling on the concept level, using constraints for expressing restrictions in a domain, and using procedural knowledge for guiding the configuration process. In this paper, we show how these methods can be used for deriving products in a software product line environment. For example, diverse types of entities like features and artifacts and their relations are modeled. Furthermore, in structure-based configuration the models have certain semantics, which enable automatic configuration. Thus, those models can be used for product derivation not only for manual examination. Experiments at our industrial partners show that these methods can in principle be applied to product derivation. Furthermore, the existence of structure-based configuration tools enables an easy application of automatic configuration.

References

1. V. Arlt, A. Günter, O. Hollmann, T. Wagner, and L. Hotz, ‘EngCon - Engineering & Configuration’, in *Proc. of AAAI-99 Workshop on Configuration*, Orlando, Florida, (July 19 1999).

2. T. Asikainen, T. Soininen, and T. Männistö, 'Representing Software Product Family Architectures using a Configuration Ontology', in *Proc. of 15th European Conference on Artificial Intelligence (Configuration Workshop)*, pp. 113–118, Lyon, France, (July 21-26 2002).
3. Joachim Bayer, Cristina Gacek, Dirk Muthig, and Tanya Widen, 'PuLSE-I: Deriving Instances from a Product Line Infrastructure', in *7th IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, pp. 237–245, Edinburgh, Scotland, (2000).
4. A. Günter, *Wissensbasiertes Konfigurieren*, Infix, St. Augustin, 1995.
5. A. Günter and L. Hotz, 'Auflösung von Konfigurationskonflikten mit Wissensbasiertem Backtracking und Reparaturanweisungen', in "*Wissensbasiertes Konfigurieren*", ed., A. Günter, St. Augustin, (1995). Infix. in german.
6. A. Günter and L. Hotz, 'KONWERK - A Domain Independent Configuration Tool', *Configuration Papers from the AAAI Workshop*, 10–19, (July 19 1999).
7. A. Hein and J. MacGregor, 'Managing Variability with Configuration Techniques', in *Proc. of the Workshop on Software Variability Management at the International Conference on Software Engineering*, Portland, Oregon, USA, (May 2003).
8. L. Hotz and T. Krebs, 'Supporting the product derivation process with a knowledge-based approach', in *Proc. of Software Variability Management Workshop at ICSE 2003*, Portland, Oregon, USA, (May 3rd 2003).
9. L. Hotz, T. Krebs, and K. Wolter, 'Dependency Analysis and its Use for Evolution Task', in *submitted to 18th Workshop, New Results in Planning, Scheduling and Design (PuK2004)*, (2004).
10. K. Kang, J. Lee, and P. Donhoe, 'Feature-oriented Product Line Engineering', In: *IEEE Software*, **7**(8), 58–65, (2002).
11. T. Krebs, L. Hotz, and K. Wolter, 'Pre-Packaged Variability for Product Derivation in Product Lines', in *Proc. of the Configuration Workshop on 17th European Conference on Artificial Intelligence (ECAI-2004)*, Valencia, Spain, (August 2004).
12. T. Männistö, T. Soininen, and R. Sulonen, 'Configurable Software Product Families', in *Proc. of European Conference on Artificial Intelligence 2000 - Workshop on Configuration*, Berlin, Germany, (August 2000).
13. Dirk Muthig, 'Implementierung von Software-Produktlinien in der Praxis', *Objektspektrum*, 66–69, (2004).
14. K.C. Ranze, T. Scholz, T. Wagner, A. Günter, O. Herzog, O. Hollmann, C. Schlieder, and V. Arlt, 'A Structure-based Configuration Tool: Drive Solution Designer DSD', *14. Conf. Innovative Applications of AI*, (2002).
15. C. Schröder, R. Möller, and C. Lutz, 'A Partial Logical Reconstruction of PLAKON / KONWERK', in *DFKI-Memo D-96-04, Proceedings of the Workshop on Knowledge Representation and Configuration WRKP'96*, ed., F. Baader, (1996).
16. T. Soininen, J. Tiihonen, T. Männistö, and R. Sulonen, 'Towards a General Ontology of Configuration', *Artificial Intelligence for Engineering Design, Analysis and Manufacturing (1998)*, **12**, 357–372, (1998).
17. M. Stumptner, 'An Overview of Knowledge-based Configuration', *AI Communications*, **10**(2), 111–126, (1997).
18. S. Thiel and A. Hein, 'Systematic Integration of Variability into Product Line Architecture Design', in *Proceedings of the 2nd International Conference on Software Product Lines (SPLC-2)*, (August 2002).