RWTH Aachen

Department of Computer Science

Technical Report

Modelling Requirements and Architectures for Software Product Lines

Horst Lichter, Thomas von der Maßen and Thomas Weiler

ISSN 0935–3232 · Aachener Informatik Berichte · AIB-2002-05

RWTH Aachen · Department of Computer Science · February 2002

The publications of the Department of Computer Science of RWTH Aachen (*Aachen University of Technology*) are in general accessible through the World Wide Web.

http://aib.informatik.rwth-aachen.de/

Modelling Requirements and Architectures for Software Product Lines

Horst Lichter, Thomas von der Maßen and Thomas Weiler

Lehr- und Forschungsgebiet Informatik III
RWTH Aachen, Germany
Email: {lichter, vdmass, thweiler}@cs.rwth-aachen.de

Abstract. The development of software product lines has become a new and promising field in software development in the last few years. Market asks for faster development of new software products which also must be cheap and of high quality. Here software product line engineering offers software companies the possibility to adress this market needs by also reducing the development costs.
Software product line engineering is based on the *domain engineering* which provides a basis of *core assets* (also called *platform*), which in turn can be reused by thereon based applications in the *application engineering*. Thereby it is essential to model the variability which occurs among different applications derived from the platform.
Within requirements engineering the specification of requirements only in natural language isn't adequate to obtain a most possible complete and consistent description of requirements. Because existing notations aren't adequat for modeling requirements for software product lines a new approach is required.
Also architecture modeling for software product lines requires new concepts for modeling the common and variable parts of a software product line. Therefore software architectures for software product lines can be described by means of *feature components* which represent a specific characteristic of the system to be modeled.
This report adresses these problems and offers new concepts for their solution.

1 Introduction

The development of software product lines has become a new and promising field in software development in the last few years. The idea behind the concept of a software product line is not to build one software product after another, but to develop a product family in that each member shares a set of commonalities with all other members and differs in particular aspects. This approach is very common in traditional engineering fields like for example the automobile industry or even the computer hardware industry.

Today most software applications are still constructed in a single-product-fashion. On the basis of customer requirements an application is created and once the application meets the customer's requirements the product is released. Because reuse is no strategic objective none or only small effort is invested in the identification and creation of possibly reusable elements which can be used in future projects. But, increased reuse is a key factor in software development to fasten the development process and to deliver products in time. The approach to develop a software product family for a particular domain based on a common shared platform attempts to address this objective.

1.1 Evolution of Reusablity Concepts

Since its early days software engineering has regarded reusability an important objective to be archieved and a central means to improve software development. For this reason increasing reusability was always a goal when developing new methods, techniques, and languages.

As shown in figure 1 there is an ongoing evolution in the development of reusabiliy concepts.

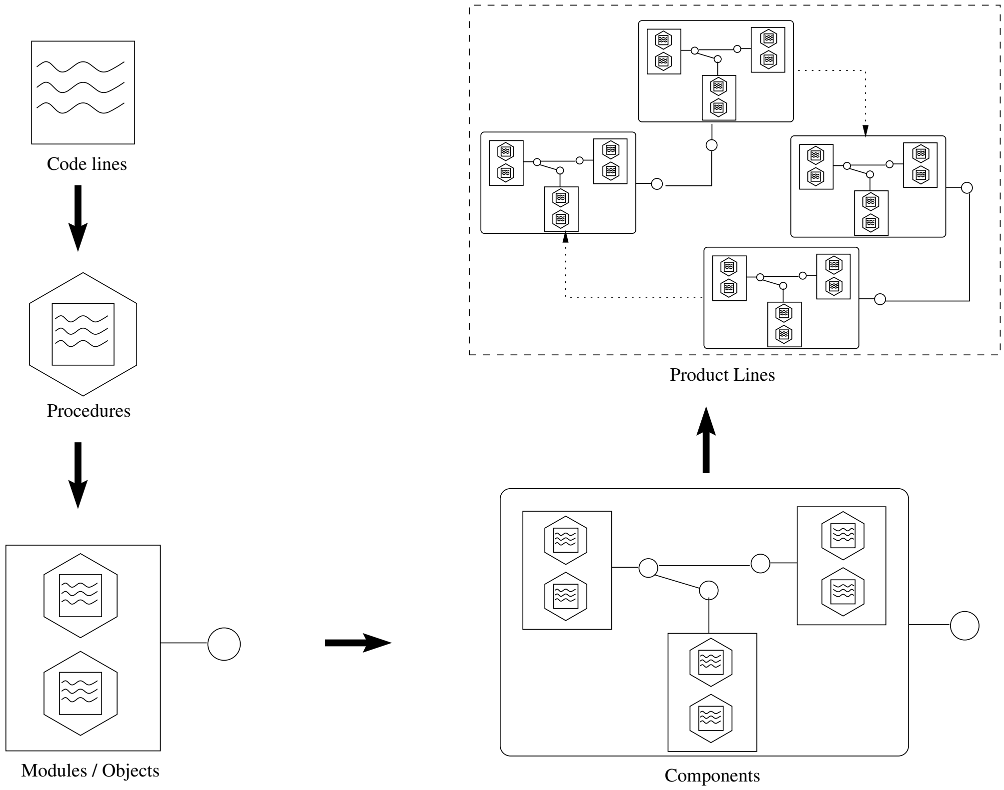

Fig. 1. Evolution of reusability concepts

- *Copy-and-Paste*
  At first programs consist of a (long) sequence of code lines without any grouping or separation. Code lines implementing a general reusable functionality were ”reused” by copy-and-paste. Hence, this code has to be maintained as often as it was copied.
- *Procedural Abstraction*
  Because the copy-and-paste approach was very ineffective and error-prone code lines implementing a task were grouped together and made availabe for reuse by the concept of *functions* and *procedures*. After a procedure or function is defined

it can be called by its name together with its actual parameters as many times as needed.

- *Modules and Objects*
  But as the size of software systems grew only procedural abstraction which still leads to monolithic systems remains unsatisfying. To develop large software systems a concept was needed to divide the whole system into smaller units which can be independently developed, tested and integrated. Each unit hides its inner structure and complexity and offers an interface for external access. In the 70th *modules* and later *objects* were introduced to implement the concept of information hiding i.e. to encapsulate data and to offer operations to manipulate this data.
- *Frameworks*
  Based on the object oriented concepts frameworks define a reusable architecture for a class of applications. A framework defines the central abstractions and design decisions for this application class. Based on a framework applications can be built by subclassing abstract classes of the framework and by providing additional classes needed to complete an application. Concerning reusability frameworks focus on both architecture and implementation reuse.
- *Components*
  Experience shows that objects, classes, and inheritance are well suited to conctruct applications and to increase reusability on a single application level. On the other side inheritance is a difficult means and leads to a very strong coupling of the classes involved. To overcome this problems, the concept of components was introduced. A component offers a set of useful and reusable functionality at its interfaces. A component can be reused in every context where its specific task must be solved. Components can be easily assembled resulting in a new application.
- *Software Product Lines*
  Software product lines are the latest and advanced concept of software reusability. The idea is not only to reuse components where similar tasks must be done but also to reuse *structures of collaborating components* to solve similar problems. This can be seen as a *framework of higher-level software elements*, which abstractly describes the relation of components in a family of similar software products. Thereby *component* doesn't necessarily mean components as used in JAVA™ Beans, CORBA™ or Microsoft™ ActiveX/DCOM but a higher-level software element, which could also be again something like a framework. Software product lines therefore introduce a reuse strategy at a still higher abstraction level as objects and components respecitvely and even frameworks.

1.2 Definitions

The Software Engineering Institute (SEI) at the Carnegie Mellon University gives the following definition of a *software product line*:

> ”A software product line is a set of software-intensive systems sharing a common, managed set of features that satisfy 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.” [SEI].

In that definition it is stressed, that the development is constrained by a prescribed way. The software products have to be built from a set of core assets instead of being built separately. A software product line is therefore a set of products that have a common core of characteristics - in this report called *platform* - and have a set of different characteristics, the *variable parts*. Figure 2 shows this context.

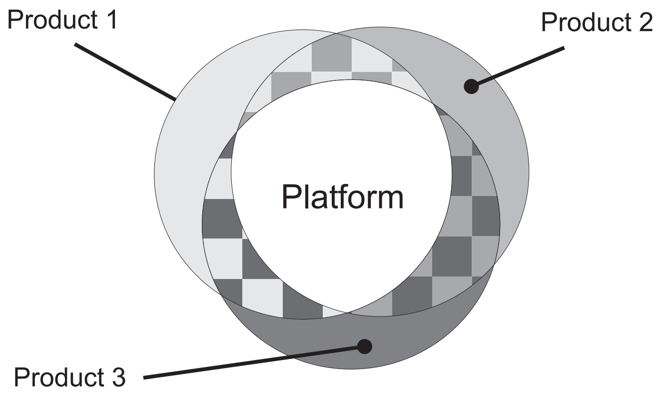

Fig. 2. Products and their shared platform

Closely related to the concept of a software product line is the concept of a *product family*. As early as 1976 David Parnas introduces this concept to the software engineering community [Par76]. Weiss et al. refine this concept in the context of software product lines [BSW99]. This refinement leads to the following stages in defining a product family:

- *Potential product family*
  A set of software for which one suspects that there is sufficient commonality to be worth studying the common aspects of the software.
- *Semi product familiy*
  A set of software for which common and variable aspects have been identified.
- *Defined product family*
  A semi-family for which an economic analysis has been performed in order to decide how much of an investment should be made in the family. This investment exploits the commonalities and variabilities for the purpose of efficiently creating family members.
- *Engineered product family*
  A defined family for which the organizational and technical infrastructure has been set up in order to develop the family members (i.e. the products).

Based on these definitions a software product line can be seen as an engineered product family where each product is launched, sold, and shipped seperately.

1.3 Expected Benefits

Reuse of high level artifacts is the main objective of the product line development approach. As stated by Jacobson et. al. [JGJ97] reuse strategies have three major goals:

- *Faster development*
  Time to market has become a crucial factor for being successful in todays software markets. Reusing artefacts leads to shorter development times because products are not build from scratch each time.
- *Improved quality*
  Only products that fit the users requirements and are of high quality are accepted and used. We expect that the reuse of validated high quality artifacts leads to better quality of the resulting new products.
- *Reduced development and maintenance costs*
  Because product development is conceptionally based on reusing high quality artifacts and products have not to be implemented from scratch the overall development costs will decrease. The same holds for the maintenance costs since only the product specific aspects must be developed and maintained separately.

By consequently applying reuse strategies in all development areas noticeable improvements can be achieved. Jacobson et al. report the following improvements [JGJ97]:

- Factor 2 to 5 faster time to market
- Factor 5 to 10 less error sensibility
- Factor 5 to 10 less maintenance costs

The overall development costs are reduced of around 15% to as much as 75% for long term projects. Furthermore, the application of reuse strategies creates highly adaptable products, consistently usable products and higher market agility.

2 Software Product Line Engineering

Because developing software product lines needs new or adapted methods and techniques a new engineering field called *Software Product Line Engineering* emerges. Software Product Line Engineering concerns with the definition of methods, techniques, tools, and processes to support software product line development. Each software product line must be carefully defined and planned. Its realization must be implemented with great disciplin. Therefore the development of a software product line puts high demands on the whole software development process. A very high level process for developing software product lines is given by the SEI. It defines the following three main parts: *Domain Engineering, Application Engineering, and Management* (see 3)

Next we will describe these major aspects of software product line development in more detail.

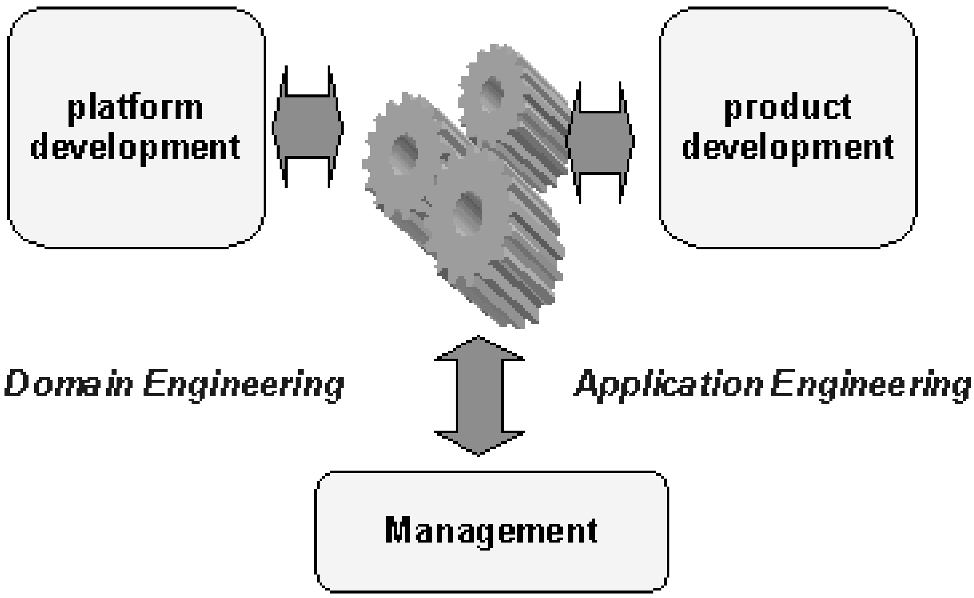

Fig. 3. High Level Product Line Engineering Process (acc. to [SEI])

2.1 Domain Engineering

By identifying common parts of similar products and major aspects of a given domain respectively a basis of shared artifacts is created. This process is called *Domain Engineering*. So the task of the domain engineering can be defined as: Identify similar structures (*features*) from a set of related products or problems of a given domain and provide a basis of elements which can be shared among thereon based applications.

This definition is consciously generally, because domain engineering doesn't only mean to provide a basis of shared software elements but also sharing similar documents or document templates, requirements artefacts or (abstract) test cases for example.

Furthermore domain engineering may mean to analyze

- a given domain of a specific problem or
- a set of closely related, already produced applications.

In the first case (the greenfield approach) the development of the platform is only based on the *domain knowledge* of the domain engineers. Therefore *domain experts* with experiences in the regarded area are needed to fulfill this task. In the latter case the common basis is created on the similarities among the analyzed products and also on base of the *domain knowledge* of the domain engineers.

The result of the domain engineering is a model defining the common platform of the product line. This *domain model* will be used by the application engineering to produce *application models* for each specific product.

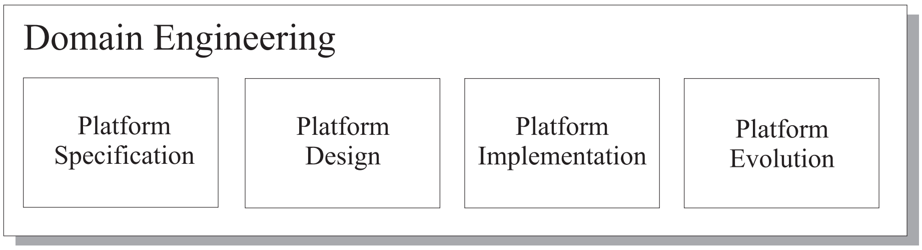

Fig. 4. Structure of the Domain Engineering Process

The domain engineering process can be divided into the activties depicted in figure 4. After the scope of the product line and the products belonging to the product line are defined, a common architecture must be developed and the platform must be implemented. The maintenance and evolution of the platform is also part of this process.

2.2 Application Engineering

In the *Application Engineering* single products are created by (re-)using the artifacts provided by the common platform (often called *core assets*), see figure 5.

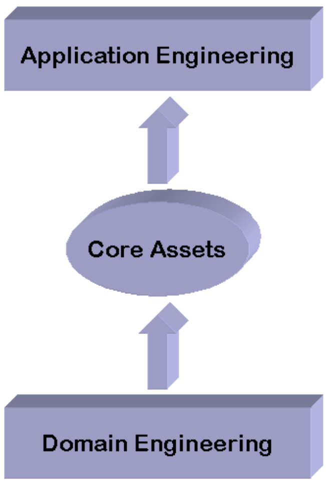

Fig. 5. Domain and Application Engineering

For that purpose the specific features of an application to be build must be specified and determined. The domain model should support the application engineer in this task by providing decision guidance which helps to map customer requirements to features provided by the platform. Features might be provided by platform or might also be special features which can - in the worst case - only be identified for a single product. In the first case features and their implementation can be (re)used in the latter case features must be newly specified and implemented. As will be shown in more detail in section 5 the domain model will also help to determine which dependencies exist between different features, so that the application engineer knows which dependent features must also be included if he decides to include a specific feature in the application. Figure 6 visualizes the main parts of the Application Engineering process.

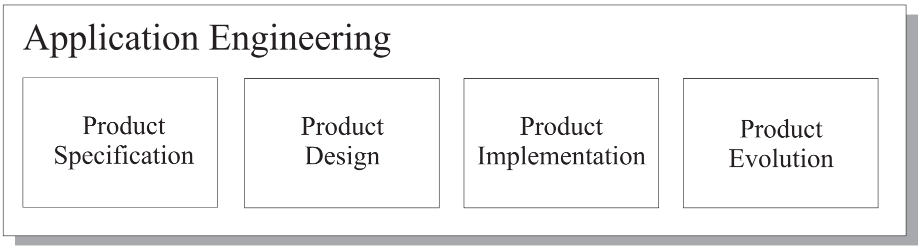

Fig. 6. Structure of the Appication Engineering

2.3 Management

The main task of management is to coordinate the two processes (domain and application engineering), because they are not completed one after the other but are highly performed in parallel.

Hence, management plays a critical role in successfully developing a product line. Activities of both domain and application engineering must be given resources, they must be coordinated, and supervised. Organizational management must set in place the right organizational structure that makes sense for the enterprise, and make sure that the organizational units receive the right resources. Organizational management is the authority that is responsible for the ultimate success or failure of the product line effort. Organizational management also contributes to the core asset list, by making available for reuse those management artifacts (especially schedules and budgets) for producing products in the product line.

The professional management of a product line development is a prerequisite to gain the benefits of this approach. Experience shows that a developing organization must have mature software development processes to move from single product development to the product line development approach. This holds especially from a management point of view.

In the following section some case studies in producing software product lines will be presented and analyzed. These approaches often differ greatly from each other because they often are an adaption of existing methods and technologies which were used in the companies before they decided to migrate to a product line based software development. Furthermore the different domains of this companies caused different solutions which were adequat for the specific company but wouldn't work for other companies in general.

3 Practical and research experiences

Within literature and practice different approaches for the introduction and implementation of a software product line can be found. In this section at first two of them shall be presented. In section 5 this different approaches will slip into a first draft version of a meta-model for the architecture of software product lines which tries to identify the modeling elements needed to describe a *domain architecture* and a thereof derived *application architecture*.

3.1 Domain-Oriented Engineering of Elevator Control Software

LG Industrial Systems Co. Ltd. (LGIS) is one of Korea's leading suppliers of elevator control systems. The diversity of customers' needs, rapidly changing market requirements and the necessity to respond quickly to the actions of market competitors forced LGIS to consider new ways in software developing. By utilizing reusable and adaptable components LGIS achieved to reduce maintenance costs drastically [LKK$^+$00]. The method used can be divided into three parts, see figure 7

– Domain analysis
– Separation of behaviour, function and implementation
– Architecture based software composition and generation

Domain Engineering Within the domain analysis commonalities and differences among a family of products in terms of *product features* are analyzed. The results are then organized into a *feature model*, which is used to develop domain models (i.a. operational models, component models and architecture models). The *feature model* only represents the static aspects of a given domain, that is, it describes *structural* and *compositional* aspects of the features. An example of a feature model of elevator control software is given in figure 8. The dynamic characteristics are modeled with the help of message-sequence diagrams (MSD), statechards and data flow diagrams (DFD). This *operational models* describe how the features cooperate within the given domain.

The features and their interaction are then modeled as components in a *component model*. The purpose of this component modeling is to develop reusable and adaptable components. Thereby the term component refers to any unit of reuse or integration, including computational components, interface components, communication components and implementation package components. In the next step components are decomposed into objects which are described by an *object model*.

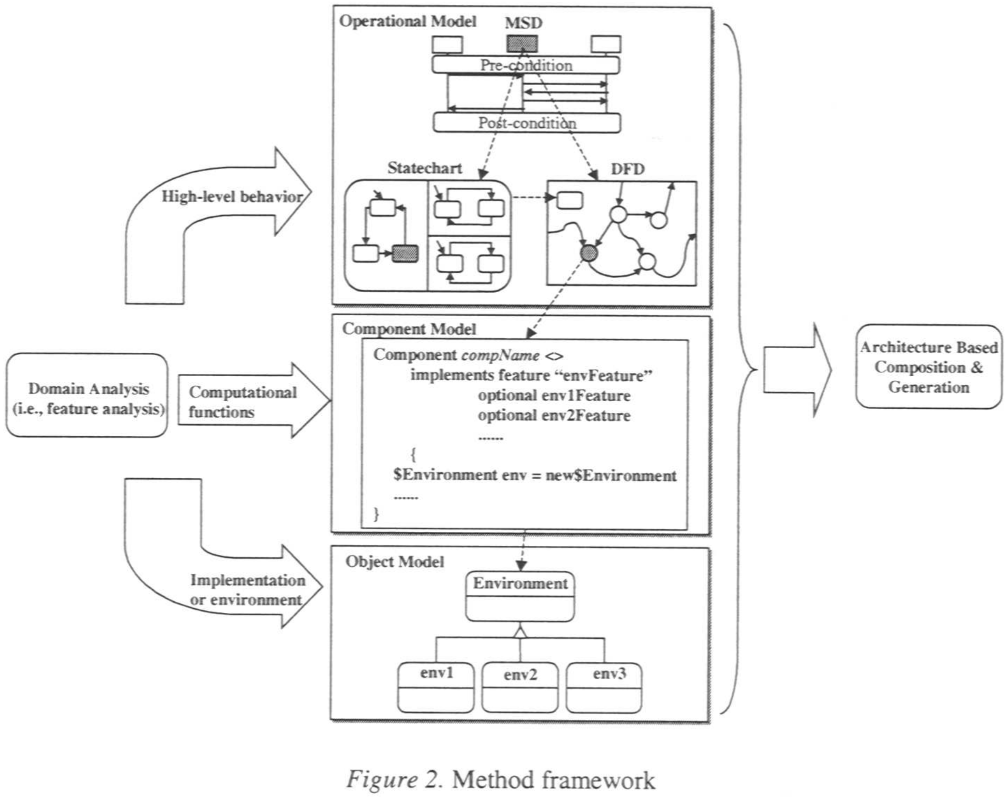

Fig. 7. Domain-Oriented Engineering at LGIS [LKK+00]

In the last step an *architecture model* is produced which maps the logical structure provided by the operational models to the physical configuration of the problem domain. This may mean to map the logical structure on a centralized or a distributed IO Control System.

Application Engineering Application engineering is a process to obtain an executable source code while utilizing models developed in the domain engineering phase [LKK+00]. This is done by selecting the necessary features for a spezific product which in turn requires to select the components needed. Based on this selection automatic code generation is used to produce executable code for the spezific application within the given architectural environment. Thereby only two JAVA classes are generated - one for the program logic and one for the data used in the program logic.

This approach surely can only be used in domains where problems can easily described by state machines - as it is the case for elevator control software - which can be implemented the way it is done in the given example by LGIS. In the next section software product line development at Cummins Engine Company will be presented. Thereby focus will be layed upon the problems which arrise in introducing and executing software product lines in a company.

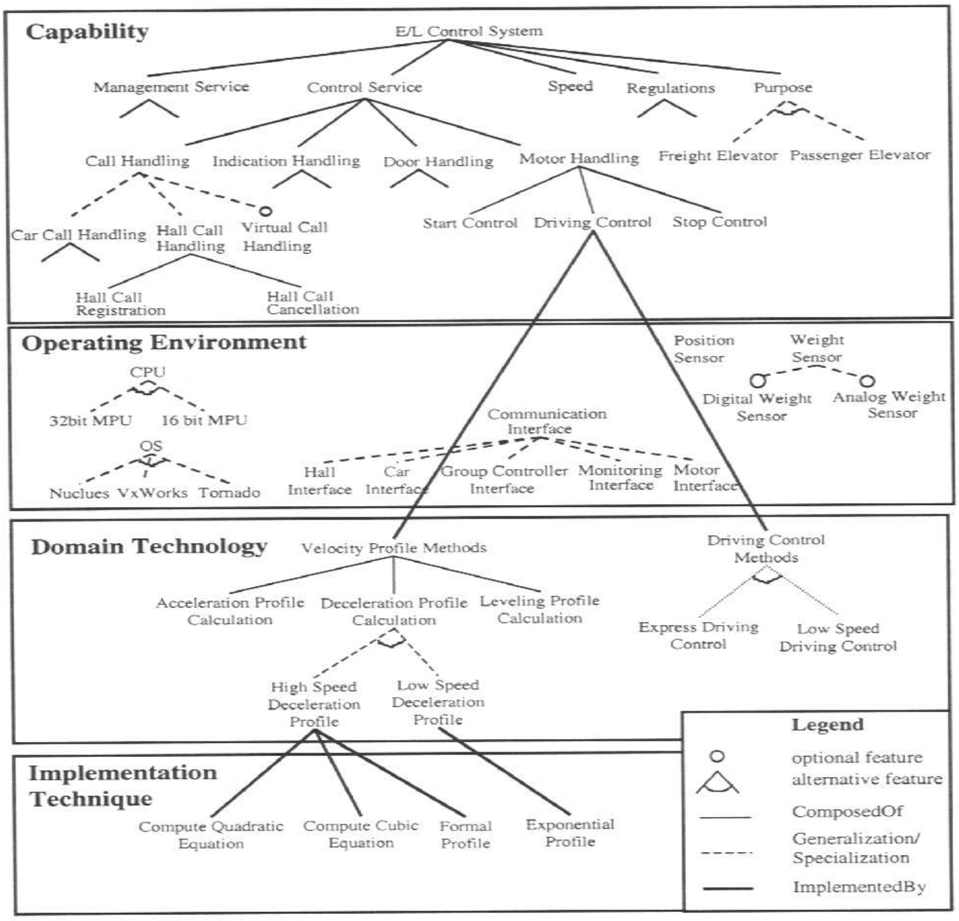

Fig. 8. Feature model of elevator control software [LKK+00]

3.2 Software Product Line Development at Cummins

Around 1994 Cummins Engine Company established a software product line for its real-time embedded diesel engine controls in cooperation with the Software Engineering Institute (SEI) of Carnegy Mellon University [Dag00]. The results of this project also sliped in the *framework for software product line practice* developed at the SEI [SEI] which tries to give an overall description of software product lines. The software product line program at Cummins slashed development costs and time to market and launched many successful products. But the introduction of software product lines wasn't without throwbacks. Five years after Cummins started to use a product line based approach in software development the overall reuse effort was reduced by 50 percent. Therefore management of Cummins decided to analyze this evolution which leads to the following results

- Components and interfaces were to complex

– Insufficient domain analysis resulted in frequently changed components which in turn caused instability of components
– Lack of training for software developers in applying the product line
– Distribution of configuration management resulting in non-synchronized repositories
– Lack of a product line supporting development process
– Insufficient documentation of requirements, architecture and design decision

Furthermore the analysis showed the necessity to concentrate of spezific parts of an overall software architecture by *separation of concern*. Consequently a lack of tools which suport different views on an architecture was stated.

Besides that reviews were introduced to improve the communication between customers and developers but also within development teams and to eliminate errors, misunderstanding and inaccuracy in early phases of the software development process. This was essential for the development of software product lines because inaccuracy in early phases of the domain analysis resulted in errors which affected every product which is based on the errorness domain model. All this change of structure resulted in the development of a completely new product line basis which is actually be used at Cummins to improve their software product line approach.

As shown in this section modeling software product lines requires specific modeling elements not found in ”conventional” software engineering. *Features* for example are used to designate a behavioural or quality characteristic of a domain and application respectively. As stated in section 1 this feature modeling is at a still higher abstraction level then components or even frameworks. These features can be grouped into *common* and *variable* features found in all respectively few applications.

Furthermore the possibility to concentrate on specific aspects of the overall system architecture is still more important for complex architectures like software product lines. In section 5 the required modeling elements for modeling software product lines should therefore be identified and structured by a meta model.

4 Requirements Engineering for Software Product Lines

4.1 Overview

In this section we will discuss the main problems of the requirements engineering process for software product lines. In section 4.2 the new topics and problems concerning the requirements engineering process will be presented. Furthermore various characteristics of a product line are mentioned and requirements for a requirements engineering model that supports software product line development will be presented. One of the main tasks in requirements engineering for software product lines is to capture variability. In section 4.3 we present requirements for a graphical notation to model variability. Further we analyse two existing notations for modelling variability in requirements and find out that none of them is able to fulfill all the requirements that have been elicitated. Finally we present a conclusion and discuss related and future work on this topic.

4.2 Requirements Engineering for Software Product Lines

The analysis, as a first step in system development, is a very important activity that mainly influences the success of a project. This is true for single product development and also (and is even more important) for product line development. The SEI mentions that requirements engineering encompasses the following processes:

- requirements elicitation: the process of discovering, reviewing, documenting and understanding the user´s needs and constraints for the system
- requirements analysis: the process of refining the user´s needs and constraints
- requirements specification: the process of documenting the user´s needs and constraints clearly and precisely
- requirements verification: the process of ensuring that the system requirements are complete, correct, consistent and clear
- requirements management: the process of scheduling, coordinating and documenting the requirements engineering activities

When developing a software product line, we have to consider additional aspects in contrast to a single product requirements engineering. These aspects are discussed in the next section.

Challenges The processes in requirements engineering for software product lines listed above must consider additional aspects in contrast to single systems engineering. It is necessary to:

- identify commonalities and variations
- model commonalities and variability
- specify platform and product requirements
- verify platform and product requirements
- provide the integration of future requirements to both, platform and products

During the requirements elicitation the specific needs of the customers, respectively users of the products must be found. These information may be expressed in marketing reports or through existing domain knowledge. During the elicitation, an explicit domain analysis is necessary. In literature the importance of scoping is well defined. Scoping is one activity during the requirements engineering process for software product lines. The product line scoping is a well defined activity in the Product Line Practice Framework, developed by the SEI [SEI]. Scoping is also an activity in the product line framework PuLSE (Product Line Software Engineering) developed by the Fraunhofer Institute for Experimental Software Engineering (IESE) [DS00]. Here, scoping is embedded in PuLSE-Eco, one technology component within PuLSE. Scoping is a critical activity, because the boundaries of the product line, that means to define which products will belong to the product line and the common parts of it are defined. Therefore the analysis of the requirements determines which requirements are platform and which are product requirements. Figure 9 illustrates this correlation.

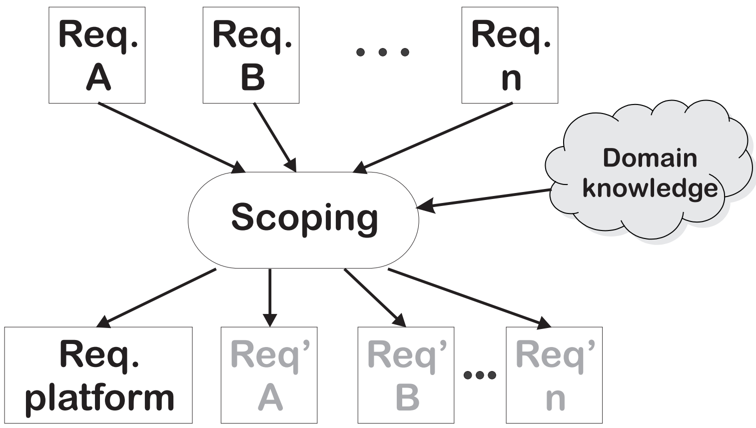

Fig. 9. Analysis of common and variable requirements for the different products

The elicitated requirements are requirements for products only (requirements for product A, B and so on). These requirements may be elicitated from customers or through marketing reports. The task of scoping is now to analyse these requirements and together with existing domain knowledge, requirements engineers have to decide which requirements are modelled in the platform and which are specific requirements for the different products. These new requirements have to be transformed and trimmed, because some of the requirements that where elicitated for product A have become platform requirements. The structure, the flexibility and the evolution of the product line will then be strictly fixed, that means, potential future products must then be fitted into an existing architecture. Besides the distinction between platform and product requirements, the variability in the different products must be modelled and verified. Furthermore it must be mentioned, that a software product line is open-ended, that means that over time there will be new requirements for potential future products. Therefore the traceability of documented requirements is extremely important.

In the next section some problems concerning the requirements engineering process for product lines will be presented.

Problems concerning the requirements engineering process The problems concerning the requirements engineering process can be divided into two categories which will be presented in the following.

Management problems As said before, professional management is cruical for successfully developing a software product line. Problems to be mastered may be e.g.

- Communication
  A lot of communication is needed to coordinate the domain and the various application engineering processes concerning requirements management. If this communication is not set up adequately and not monitored, problems in identifying and classifying of requirements will arise.
- Right requirements classification
  During platform and product maintenance new requirements must be integrated properly. Often it is not not clear, whether new requirements should be integrated in the platform or are specifc for one product. Wrong decisions regarding this requirements classification lead often to implementing requirements in products, which belong obviously into the platform. These decisions must be made explicitly by a change control board that is responsible for all products of a product line.
- Measuring benefits
  A serious problem is to value the costs and the benefits of the platform development. This can be done be regarding the return of investment for a product line development, i.e. defining at which point the large expense at the beginning of the product line development will be leveled and the whole project becomes profitable.

Methodical problems Methodical problems cover problems concerning methods and notations to model common and variable aspects of the product line. We will see in section 5.3, that there is no special method or notation to express variability in requirements. Therefore existing notations are taken and modified to suit to the needs of the requirements engineer but until now, no notation is applicable to fulfill all the requirements that were put on such a notation.

Another important aspect is the lack of suitable tools supporting the requirements engineering activities for product lines. That means tools to model and trace platform and product requirements as well as commonalities and variations.

4.3 Modelling variability

With respect to the methodical problems mentioned above, a notation is required, that is able to express variability. Variability expresses that a part of the system deviates in a defined way. So called variation points can be defined. In these points different, concrete variants can be chosen to resolve this variation point. For example we want to consider to administer bank accounts in a homebanking-system. At these variation point it is possible to administer giro accounts and/or fixed deposit accounts, whereas it should always be possible to manage giro accounts. Examining this simple example, it is necessary to distinguish between different types of variability and relationships between variations. The following types of variability must be considered:

- options

– alternatives

Optional aspects of a system can be integrated or not. That means from a set of optional aspects, any quantity of these aspects can be choosen, including none or all. Hence, so optional aspects can be modelled by means of an or-relatonship. From a set of alternative aspects, only one aspect can be choosen - defining an exclusive-or (xor)-relationship.

Besides the distiction of the different types of variability, the relationships between the variable parts must be defined, too. These relationships define constraints with respect to the choice of variant parts. The following constraints are needed to express the different relationships:

– implied
– equivalent
– mutual exclusiv

The implied-relationships (A, B) expresses, that if aspect A is choosen the implied aspect B must be choosen, too, whereas an equivalent-relationship is an implied-relationship in both directions. An exclusiv-relatonship (A, B) expresses, that if aspect A is chosen, it is not allowed to choose aspect B as well, and vice versa.

To express variability it is necessary to model *variation points*. A variation point is a location within the system where functionality differs in some way. This differentiation will be examined later in this section. In the first place it is necessary to stress the importance to supplement natural-language requirements specifications with semi-formal notations. By the majority, in practice requirements specification are composed in natural language. In spite that it is possible to write correct, clear and sound requirements in natural language [Rup01] it is very difficult to express variability. Another point is, that the various products have to be discussed with domain experts, respectively the customers in a way that both sides can understand what the potential systems should be able to perform.

Requirements for a notation to model variability As usual notations can be graphical a textual or a mixture of both. The advantage of modelling variability in a graphical notation is, that variation points are recognized much easier than in a pure textual description. In this section we want to present the requirements that have to be put on a notation that provides to express variability in Software Product Lines. The following requirements have been identified:

– representation of common and variable parts
– distinction between types of variability
– representation of dependencies between variable parts
– providing different views
– possibility to add future requirements easily

– providing good tangibility for domain experts and system developers

Representation of common and variable parts. In a graphical representation it must be possible to express common and variable parts of the system. That means there is a need of a graphical element to express aspects that belong to the platform and therefore shared by every product of the product line. Another element is needed to express variations at a specific point.

Distinction between types of variability. The notation must be able to express different types of variable parts. Types of variability comprises that there are optional and alternative parts. Optional means that these aspects may belong or may not belong to a system, whereas between two or more alternative aspects an exclusive-or relationship exists.

Representation of dependencies between variable parts. The description of dependencies is mandatory. Dependencies between variable parts are implied, equivalent and xor-relationships. An implied-relationship means, that if one aspect is needed, than another aspect must be taken into the system as well. An equivalent-relationship is an implied-relationship in both directions and a xor-relationship expresses that only one aspect from a set of aspects can be taken into the system.

Providing different views. Using the notation it should be possible to model at least two different views on the product line. It must be possible to model the product line as a whole, comprising the common and the variable parts - the platform and the various products. The representation of only one product must be mandatory, too, taking into account that one product consists of parts taken from the platform and parts that are specific for this one product.

Possibility to add future requirements. Future requirements must be added easily, that means without changing the structure of the existing parts of the system, because new requirements will arise for future products of the product line.

Providing good tangibility. In most cases domain experts do not want or are not able to understand formal specifications. On the other side, they do not want to read unstructured natural language specification, but want to get a first view of the system. So a graphical representation might help to understand the relevant parts very easily.

It is important to mention that the graphical notation should not replace natural language specifications but to supplement them and to provide a different view on the same context (see figure 10). It is mandatory to abstract from conrete requirements. The combination of requirements specified in natural language and a graphical representation should be supported by a capable requirements-template [Rup01,RR00]. This template should be modified in the way that it provides sections for platform and product requirements.

In the next section we analyse two notations with regard to the listed requirements on such a notation:

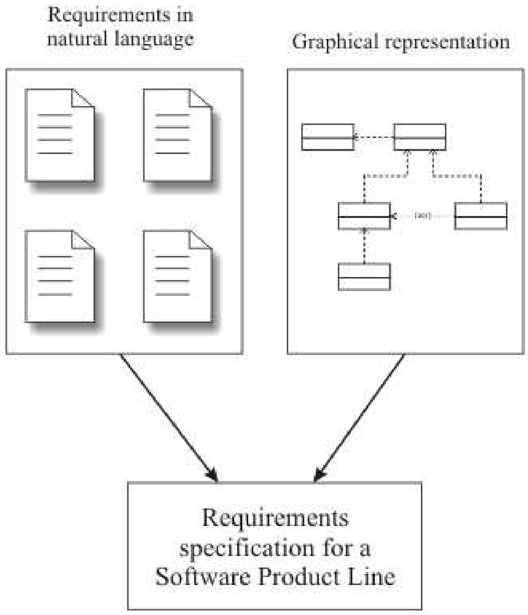

Fig. 10. Combination of requirements, written in natural language and a graphical representation of the product line

- Feature graphs from the Feature-Oriented Domain Analysis (FODA)
- Use-Case diagrams from UML

Feature graphs Feature graphs are used in the Feature-Oriented Domain Analysis method, developed by Kyo C. Kang et. al. [Kan00]. In their work they describe "a method for discovering and representing commonalities among related software systems". "The primary focus of the method is the identification of prominent or distinctive features of software systems in a domain". They define a *feature* as "a prominent or distinctive user-visible aspect, quality or characteristic of a software system or systems". Figure 11 shows an example of a typical feature graph in the context of a homebanking-system.

A filled circle indicates that a feature is mandatory, whereas an empty circle indicates that the feature is optional that means it can be integrated into the system or not. In this example, the system must provide the functionality to administer giro accounts, whereas an optional feature is to administer fixed deposit accounts that can be integrated into a product or not. An alternative between two features is captured in the way that an arc is drawn through the children of a parent node that are alternatives. The security is either handled through the Home Banking Computer Interface (HBCI) or through the PIN/TAN procedure. A line from a parent node to a child node indicates that the child feature cannot exist without the parent feature.

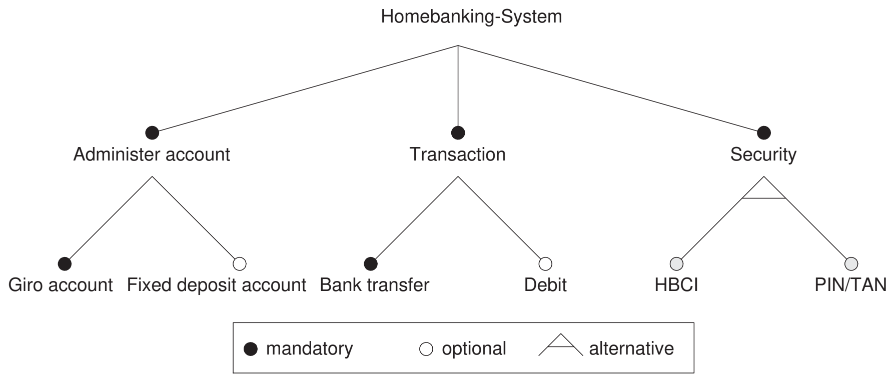

Fig. 11. Feature graph from FODA

Evaluation
The feature graphs provide a good understandable representation of common and variable parts. Common features, which are platform candidates can be identified trough a filled circle. Variable features are divided into optional and alternative features. The usage of the feature concept abstracts additionaly from concrete requirements [SGB00] and new features - generated from new requirements can be integrated easily.

Besides these advantages the feature graphs do not meet all the requirements. Dependencies between features can only be modelled implicitly through alternatives and parent-child relationships. It is not possible to model equivalent relationships or xor-relationships between features which are not alternatives. Furthermore, platform features may be scattered over the whole diagram. That makes it difficult to recognize the common parts directly.

Use Case diagrams The second notation we have examined, are the Use Case diagrams. These diagrams provide the possibility to model Use Cases. A Use Case describes a sequence of events, initiated by the user, here called actor, of the system. Furthermore it describes the interactions between the actor and the system. Use Case diagrams represent the textual target-state description of Use Cases in a graphical notation.

In these diagrams it is also possible to model relationships between various Use Cases. Use Cases can use other Use Cases to fulfill their duty. Through the inheritance-relationship, a super-Use-Case can be declared as *extension points* and the sub-Use-Cases define the actions that are embedded in the super-Use-Case. That means one Use Case can be embedded in another Use Case through an *extends*-relationship [Jac92]. Figure 12 shows an example of a Use Case in the homebanking context.

In this example, the use case ”Define access” is defined. The two sub-Use Cases ”HBCI” and ”PIN/TAN” define the steps the user must perform in the homebanking-system to install the process, provided by his bank. The Use Case diagrams provide no

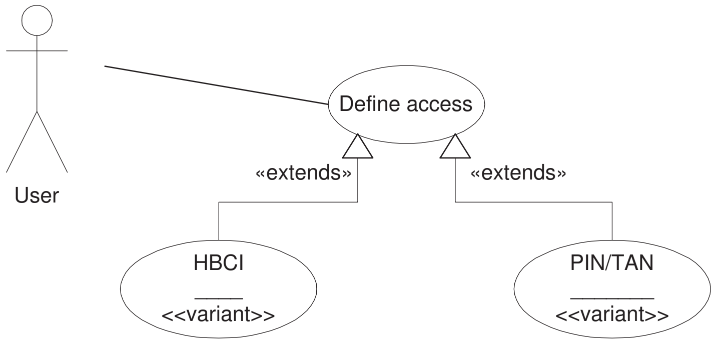

Fig. 12. Use-case diagram

graphical element to model variability. Therefore UML elements can be extended with *stereotypes* [Boo94]. Stereotypes are a powerful ability to add a meta-classification to elements with additional semantic attributes. In figure 12 the variable parts are tagged with the stereotyp ≪*variant*≫. This stereotyp defines the Use Cases "HBCI" and "PIN/TAN" as variants to the extension point of the "Define access"-Use Case. This stereotyp is insufficient to express all the types of variability presented in section 4.3. The following stereotypes will be needed to express variability adequately:

- ≪*optional*≫
- ≪*alternative*≫

Evaluation

Use Case diagrams have their advantages in modelling interactions between an actor and the system through Use Cases. The functionality of an Use Case can simply be extended through an extends-relationship to other Use Cases. The powerful ability to define new model elements through tagged stereotypes makes is easy to express variable and alternative Use Cases.

A disadvantage while using Use Case diagrams is that some relationships between Use Cases cannot be modelled. Because of that, it is not possible to define implied-, equivalent- and exclusive-relationships. Furthermore it is difficult to spot the Use Cases that are shared by all products and to identify the various products that can be build.

Conclusion In this section we specified requirements for a notation to express variability in requirements. We expressed the requirements that are essential without pretending to have a sound list of all requirements that have to be put on such a notation. We stressed further, that a graphical notation should not replace natural language specifications but should supplement them. We analysed two existing notations with respect to the required aspects. An overview of the results presented in this section is summarized in tabular 1. A "+" indicates, that the notation provides a good capability to fulfill the requested requirement. An "O" indicated a medium capability and a "-" indicates that the notations has only poor supply.

	Feature graphs	Use Case diagrams
Representation of common and variable parts	+	O
Distinction between types of variability	+	O
Representation of dependencies between variable parts	O	-
Providing different views on product line and products	+	+
Possibility to add future requirements easily	+	+
Providing good tangibility for domain experts and developer	O	O

Table 1. Comparison of feature graphs and Use Case diagrams

Both notations have their advantages: Both do abstract from concrete requirements and are able to express the requested variability. Both are well understandable and it is possible to integrate new requirements very easily. With the elements of the notations it is possible to model a product line and a single product view. On the other hand, both notations lack on expressing dependencies between variable and common parts directly. The feature graphs comprises dependencies indirectly whereas the Use Case diagrams are not able to express relationships, besides use- and extends-relationships, at all.

4.4 Conclusion and future work

Conclusion In this section we gave an overview of the requirements engineering process in the context of Software Product Lines. In section 4.2 we expressed the new aspects that arouse while implementing the requirements engineering process for product lines and pointed at the problems that will additionally come up in that process. One of the main tasks in documenting requirements for a product line is to model the common parts, shared by all potential products and the variable parts, that belong to a specific variant. There is indeed a demand for a graphical, semiformal notation to express these parts to get an better overview of what the products should perform. In section 4.3 we put on several requirements on such a notation. Furthermore we analysed two existing notations, the feature graphs from the Feature-Oriented Domain Analysis and the Use Case diagrams from the UML. Both notations have their advantages and disadvantages and both are not capable to fulfill all the requirements that were put on.

Related work The following list gives an overview of related work on requirements engineering for Software Product Lines and Domain engineering:

1. Sofware Engineering Institute (SEI). It offers the Framework for Product Line Practice and presents the Feature Oriented Domain Analysis.
   Available at: http://www.sei.cmu.edu

2. Fraunhofer Institut Experimentelles Software Engineering (IESE). The institute presents the PuLSE-framework. PuLSE provides a complete framework that covers the whole software product line development life cycle, including reuse infrastructure construction, usage, and evolution. PuLSE-Eco and PuLSE-CDA are the components of the framework that deal with economic scoping and customizable domain analysis.
   Available at: http://www.iese.fhg.de
3. TU Ilmenau Fachgebiet Prozessinformatik - Projekt Alexandria. The project work aims towards a general method for developing system families, which will integrate existing methods.
   Available at: http://www.theoinf.tu-ilmenau.de/˜pld/pub/index.html
4. Chair for Software Systems Engineering - University of Essen. The institute researches in requirements engineering for product lines using Use Cases.
   Available at: http://sse.informatik.uni-essen.de/

Furthermore, the GI-Arbeitskreis ”Requirements Engineering für Produktlinien” of the GI Fachgruppe 2.1.6 is analyzing selected industrial Software Product Line scenarios in order to determine special problems in different contexts and to recognize some kind of problem pattern and to find solutions to those problems. A final report describing the results will be published soon.

Future work The current studies deal with the enhancement of the feature graphs and the change of the UML-Metamodel to express variability directly and not through the usage of stereotypes in UML. Therefore new elements are needed to express the different types of variability and the relationships between them. The objective is to present a notation that fulfills the demands to model variability in software product lines. In future studies the integration of representing variability of requirements into requirements templates will be analysed. The objective is to support the requirements engineer in documenting the requirements of the platform and of the various products.

Concurrently a diploma thesis deals with the applicability of existing cost estimation models on the development of Software Product Lines. The goal of this work is to analyse at which points, existing methods are not able to take into account the variability of system families, like in Software Product Lines.

5 Architecture Modeling

The task of architecture modeling is to create a structure plan or a blueprint for the system to be build. The architecture of a software system can be split into two areas

– Static structure
– Dynamic structure

The static structure describes which elements of the system to be build are in relationship with other elements of the system. The dynamic structure describes how these elements interact to realize the system's functionality. Therefore the static structure can be called the *Who* whereas the dynamic structure describes the *How* of the system. Although current praxis of modeling software architectures often focuses solely on the static structure as represented by for example UML class diagrams, both aspects, static and dynamic structure, are essential for software architectures and should therefore taken into account when constructing a metamodel for SPL architectures.

Architecture modeling for software product lines can be seen as a four layer model as presented in Figure 13. The top level *SPL Architecture Metamodel* describes the

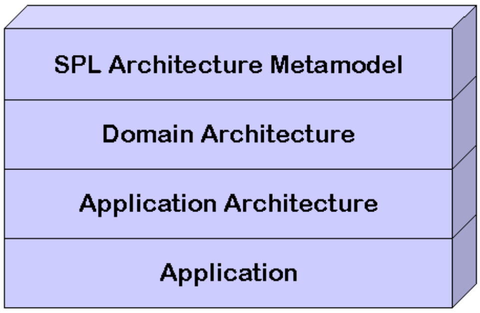

Fig. 13. Four layer model for software product line architectures

modeling elements needed to build models of the underlying level, the *Domain Architecture*. These modeling elements will be described in more detail in section 5.2. The Domain Architecture describes the overall structure of thereon based *Application Architectures* and provides the designer with customization possibilities among which he has to choose to build a concrete Application Architecture which is in turn the construction plan for a ready build product, an *Application*.

5.1 Feature Components

The central building blocks of the Domain and Application Architecture are *Feature Components*. A feature component can be seen as a self-contained unit, which represents a specific characteristic of the system to be modeled. They are an adaption of the *feature* concept introduced by the *Feature Oriented Domain Analysis (FODA)* [Kan00] to the level of architecture modeling for software product lines. It must be mentioned that the feature components at the level of architecture modeling aren't necessarily identical to the features identified at the level of the domain analysis. For example it might be possible that a set of features identified in the domain analysis together build a feature component at the architecture modeling level. Furthermore feature components need not to be realized as components provided by for example Corba or Microsoft COM/ActiveX.

The feature components can be divided into three different types as shown in figure 14

- **Common Feature Components**: They are used in a domain architecture and describe components which can occur in every application based on this domain architecture. Common Feature Components occur in derived application architectures without modification.
- **Variable Common Feature Components**: These are Common Feature Components which can occur in every derived application architecture only by specifying the variability offered by the component.
- **Specific Feature Components**: They are special building blocks needed to construct a specific architecture derived from a domain architecture.

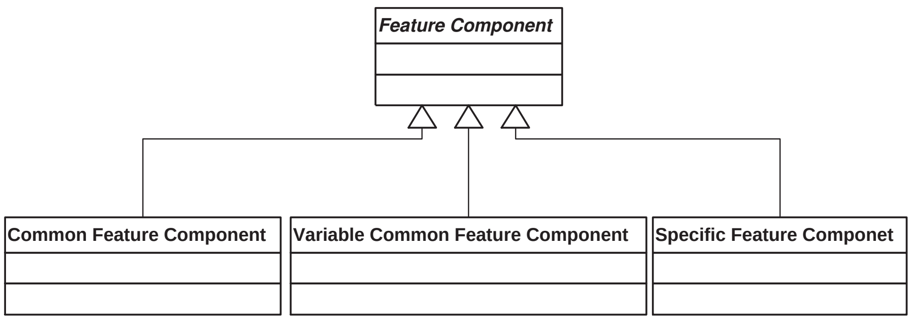

Fig. 14. Feature Components

The possible variations which can occur at Variable Common Feature Components will be described in detail in the following section.

5.2 SPL Architecture Metamodel

In this section a first draft of a SPL Architecture Metamodel will be given, see figure 15. The Domain Architecture describes the overall structure of applications which are based on it. It consists of a set of *Common Feature Compontens* which were shared among the applications, a set of *Variable Common Feature Components*, see section 5.1, and a set of *Relations* which interconnect the different feature components.

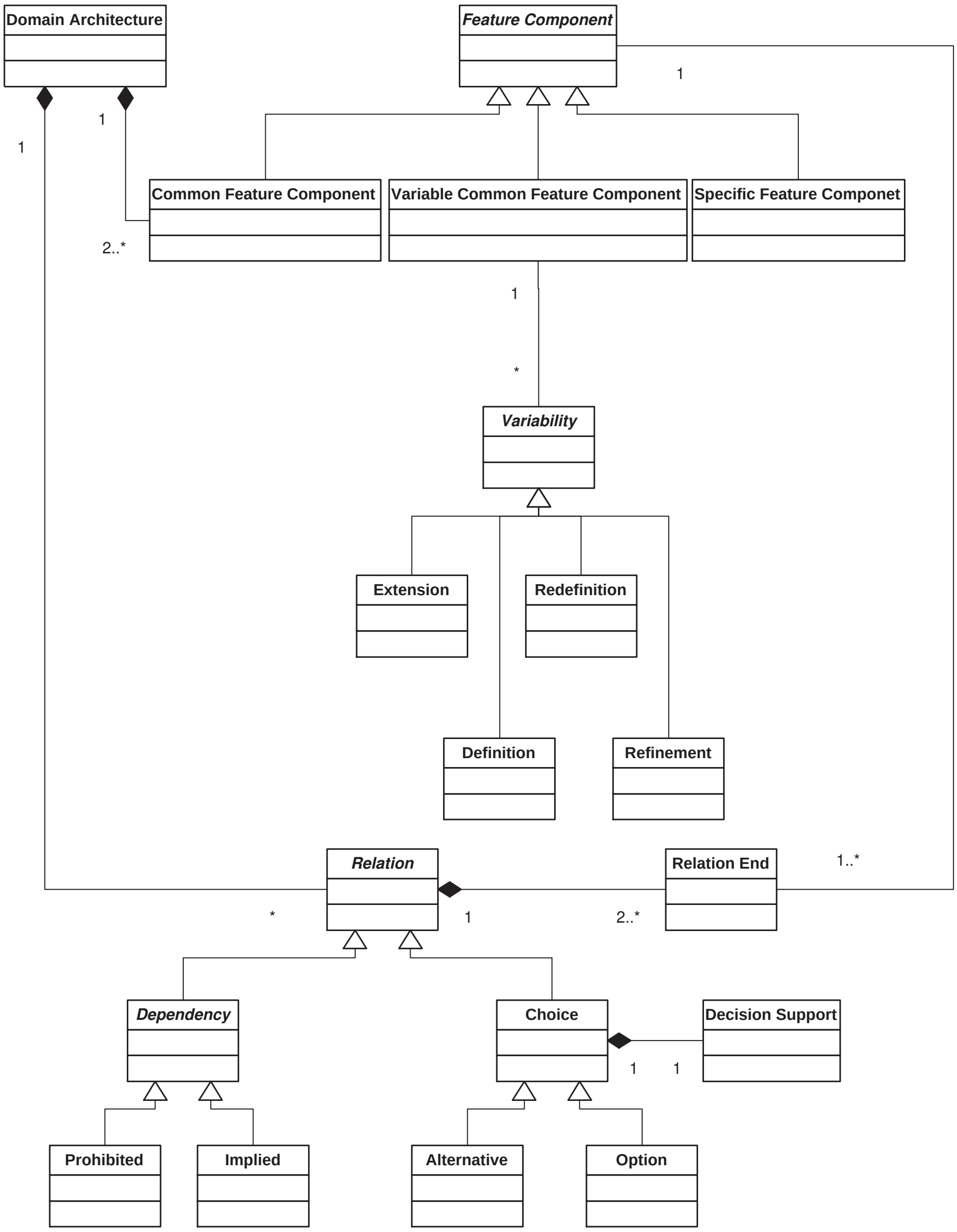

Fig. 15. SPL Architecture Metamodel

The different types of feature components were presented in the last section. It was mentioned that *Variable Common Feature Components* differ from *Common Feature Components* by offering the possiblity to modify the feature component. This variability can occur in four different ways:

- **Extension**: A feature component must be *extended* with regard to behaviour in a derived application architecture
- **Definition**: A feature component must be *defined* with regard to behaviour in a derived application architecture
- **Redefinition**: A feature component must be *redefined* with regard to behaviour in a derived application architecture
- **Refinement**: A feature component must be *refined* with regard to behaviour in a derived application architecture

Every Variable Common Feature Component includes a *note* (written in natural language) which helps the designer in processing the different types of variance.

Feature Components are interconnected with the help of relations. Thereby each relation consists of a set of *relation ends* where each end is connected to exactly one feature component. The relations can be devided into two types

- **Choice**: A choice allows the designer to choose a - possibly empty - set of feature components among a set of offered feature components. Therefore the choice provides a *decision support* which helps the designer to come to a decision. The choice can further be refined into two additionaly different kinds:
  - **Alternative**: The designer has to choose exactly one feature component among a set of offered feature components.
  - **Option**: The designer has to decide to choose a feature component or not.
- **Dependency**: A set of feature components, where the existence of one feature component depends on another feature component. This abstract type can be refined into two concrete types
  - **Prohibited**: A *prohibited* dependency between two feature components A and B means that the existence of feature component A forbids the existence of feature component B in a derived application architecture and vice versa.
  - **Implied:** An *implied* dependency from feature component A to feature component B means that the existence of feature component A enforces the existence of feature component B. That is feature component B cannot exist without the existence of feature component A in an derived application architecture.

5.3 Application Architecture

Every application architecture which is based on a domain architecture must provide an instantiation for the variable feature components which occur in the domain architecture and a choice among the feature components offered by a *choice* in the domain architecture. Furthermore each application architecture may include a set of *Specific*

Feature Components, which describe feature components not included in the domain architecture but which are possibly shared among several applications. Therefore the application architecture may also include - with respect to the domain architecture - new relations between the Specific Feature Components and the Common Feature Components found in the domain architecture and between the Specific Feature Components themselves.

Thereby the differentiation in Common Feature Components and Specific Feature Comonents is not unchangeable. Within the evolution of a software product line a Specific Feature Component can become a Common Feature Component. This may lead to a growing complexity of the domain architecture where *separation of concern* becomes more and more important. This is among others one aspect which should be considered in the development of a metamodel for software product line architectures as will be shown in the next section.

5.4 Future Work

In this section some problems in modeling software product line architectures should be presented which require future research work. At first the term *Feature Component* must be stated more precisely. It must be examined how to find the edges of a Feature Component. Furthermore an adequat notation and semantic for the definition of Feature Components must be given.

Furthermore the metamodel presented in section 5.2 should be proofed by means of a (not to complex) example to validate the identified modeling elements and to identify possibly missing modeling elements.

In addition a way to model collaborations of Feature Components must be given. Thereby modeling of static structure as well as dynamic behaviour must be possible. Because these collaborations tend to be very complex a mechanism to focus on specific parts of an overall system (*separation of concern*) must be developed.

Moreover the evolution of software product lines requires the possibility to insert (Specific) Feature Components in the set of Common Feature Components and therefore in the domain architecture. Furthermore it is necessary to trace the evolution of a software product line in order to trace design decisions.

Finally existing modeling (quasi) standards like the UML [OMG01] should be analyzed whether respectively to which level they fulfill the requirements for modeling software product lines and where already existing concepts must be extended.

6 Conclusion

As shown in this report, single-product development is not always an adequate way in software development. Market asks for faster development of new software products which also must be cheap and of high quality. Here software product line engineering offers software companies the possibility to adress this market needs by also reducing the development costs. Therefore the high demands on software companies, which are conditional upon more and more growing and changing customer requirements and also a more and more narrow market, can be fulfilled nearly without loss of earnings.

Software product line engineering is based on the *domain engineering* which provides a basis of core assets (platform), which in turn can be reused by thereon based applications in the *application engineering*. Thereby it is essential to model the variability which occurs among different applications derived from the platform. As was shown in sections 4 and 5 this is crucial for requirements engineering as well as for modeling software architectures for software product lines.

Within requirements engineering the specification of requirements only in natural language isn't adequate to obtain a most possible complete and consistent description of requirements. Because existing notations for requirements like FoDA and Use Cases don't fullfil all the requirements mentioned in section 4, like for example modeling dependencies between features, a new approach is required.

In section 5 a draft metamodel for architecture modeling for software product lines was given. This metamodel describes software architectures for software product lines by means of *feature components*. A feature component can be seen as a self-contained unit, which represents a specific characteristic of the system to be modeled. Thereby the feature components are differentiated into *common feature components*, *variable common feature components* and *specific feature components*. The feature components are in relationship among one another whereas the existence of one feature component may depend on the the existence of another feature component (*Dependency*). Furthermore a domain architecture may provide the possibility to choose a set of needed feature components among a set of offered feature components (*Choice*).

Future work will focus on the development of an adequate notation for the description of requirements as well as for architecture modeling for software product lines. Therefore existing quasi standards like the UML should be analyzed and adapted where necessary to fulfill the requirements for a usefull notation as stated in this report.

7 Glossary

Application: A ready build software product.

Application Architecture: A construction plan for an *application*.

Architecture: A construction plan. Used as synonym for *model*.

Common Feature Component: A *Feature Component* which can occur in every derived application architecture without modification.

Core Asset : A software artifact that is used in the production of more than one product in a product line A core asset may be an architecture, a software component, a process model, a plan, a document or any other useful result of building a system.

Domain Architecture: A construction plan for the overall structure of *applications* which are based on it.

Feature Component: A self-contained unit, which represents a specific characteristic of the System to be modeled.

Metamodel: A *model* which describes the modeling elements needed to build a *model*.

Model: A construction plan, see also *architecture*.

Platform : Core software asset base that is reused across systems in the product line.

Product Line : A group of products sharing a common, managed set of features that satisfy specific needs of a selected market or mission area.

SPL Architecture Metamodel: A model which describes the modeling elements needed to build a *domain architecture*.

Variable Common Feature Component: A *Common Feature Comonent* which can occur in every derived application architecture only by specifying the variability offered by the component.

References

[BCS00] D. Batory, Rich Cardone, and Y. Smaragdakis. *Software Product Lines - Experience and Research Directions*, chapter Object-Oriented Frameworks and Product Lines, pages 227–247. Kluwer Academic Publishers, 2000.

[Boo94] Grady Booch. *Objektorientierte Analyse und Design*. Addison-Wesley Verlag, 1994.

[BSW99] W Bartussek, P. Strooper, and D. Weiss. *Defining Software Families*, pages 35–40. Dagstuhl-Seminar-Report: 230, 1999.

[Dag00] J.C. Dager. *Software Product Lines - Experience and Research Directions*, chapter Cummins's Experience in Developing a Software Product Line Architecture for Real-time Embedded Diesel Engine Controls, pages 23–45. Kluwer Academic Publishers, 2000.

[DS00] Jean-Marc DeBaud and Klaus Schmidt. A systematic approach to derive the scope of software product lines. Fraunhofer Institute for Experimental Software Engineering (IESE), Kaiserslautern, 2000.

[GJC00] Cristina Gacek, Jean Jourdan, and Michel Coriat, editors. *Proceedings of Product Line Architecture Workshop, The First Software Product Line Conference (SPLC1)*. Fraunhofer IESE, August 2000. IESE-Report No. 053.00/E, Version 1.0.

[Jac92] Ivar Jacobson. *Object-Oriented Software Engineering - A Use Case Driven Approach*. Addison-Wesley Verlag, 1992.

[JGJ97] Ivar Jacobson, Martin Griss, and Patrik Jonsson. *Software Reuse, Architecture, Process and Organization for Business Success*. Addison Wesley, 1997.

[Kan00] Kyo et al. Kang. Feature-oriented domain analysis (foda) feasibility study. Technical report, SEI, Carnegie Mellon University, Pittsburgh, PA, 2000.

[KN00] T. Kishi and N. Natsuko. *Software Product Lines - Experience and Research Directions*, chapter Aspect-Oriented Analysis for Product Line Architecture, pages 135–145. Kluwer Academic Publishers, 2000.

[KS00] Peter Knauber and Giancarlo Succi, editors. *Proceedings of Software Product Lines: Economics, Architectures, and Implications*. Fraunhofer IESE, June 2000. IESE-Report No. 070.00/E, Version 1.0.

[LKK+00] K Lee, K.C. Kang, E. Koh, W. Chae, B. Kim, and B. Choi. *Software Product Lines - Experience and Research Directions*, chapter Domain-Oriented Engineering of Elevator Control Software - A Product Line Practice, pages 3–22. Kluwer Academic Publishers, 2000.

[OMG01] OMG. Unified modeling language specification, version 1.4. Technical report, OMG, 2001.

[Par76] D.L. Parnas. On the design and development of program families. IEEE Trans. on SE, SE-2:1-9, March, 1976.

[RR00] James Robertson and Suzanne Robertson. Volere requirements specification template. Atlantic System Guild www.systemguild.com, 2000.

[Rup01] Chris Rupp. *Requirements-Engineering und -Management*. Hanser Verlag, 2001.

[SEI] SEI/CMU. A framework for software product line practice - version 3.0. http://www.sei.cmu.edu.

[SGB00] Mikael Svahnberg, Jilles van Gurp, and Jan Bosch. On the notion of variability in software product lines. University of Karlskrona/Ronneby, Department of Software Engineering and Computer Science, 2000.

Aachener Informatik-Berichte

This is a list of recent technical reports. To obtain copies of technical reports please consult http://aib.informatik.rwth-aachen.de/ or send your request to: Informatik-Bibliothek, RWTH Aachen, Ahornstr. 55, 52056 Aachen, Email: `biblio@informatik.rwth-aachen.de`

95-11 *	M. Staudt / K. von Thadden: Subsumption Checking in Knowledge Bases
95-12 *	G.V. Zemanek / H.W. Nissen / H. Hubert / M. Jarke: Requirements Analysis from Multiple Perspectives: Experiences with Conceptual Modeling Technology
95-13 *	M. Staudt / M. Jarke: Incremental Maintenance of Externally Materialized Views
95-14 *	P. Peters / P. Szczurko / M. Jeusfeld: Business Process Oriented Information Management: Conceptual Models at Work
95-15 *	S. Rams / M. Jarke: Proceedings of the Fifth Annual Workshop on Information Technologies & Systems
95-16 *	W. Hans / St. Winkler / F. Sáenz: Distributed Execution in Functional Logic Programming
96-1 *	Jahresbericht 1995
96-2	M. Hanus / Chr. Prehofer: Higher-Order Narrowing with Definitional Trees
96-3 *	W. Scheufele / G. Moerkotte: Optimal Ordering of Selections and Joins in Acyclic Queries with Expensive Predicates
96-4	K. Pohl: PRO-ART: Enabling Requirements Pre-Traceability
96-5	K. Pohl: Requirements Engineering: An Overview
96-6 *	M. Jarke / W. Marquardt: Design and Evaluation of Computer–Aided Process Modelling Tools
96-7	O. Chitil: The ς-Semantics: A Comprehensive Semantics for Functional Programs
96-8 *	S. Sripada: On Entropy and the Limitations of the Second Law of Thermodynamics
96-9	M. Hanus (Ed.): Proceedings of the Poster Session of ALP'96 — Fifth International Conference on Algebraic and Logic Programming
96-10	R. Conradi / B. Westfechtel: Version Models for Software Configuration Management
96-11 *	C. Weise / D. Lenzkes: A Fast Decision Algorithm for Timed Refinement
96-12 *	R. Dömges / K. Pohl / M. Jarke / B. Lohmann / W. Marquardt: PRO-ART/CE* — An Environment for Managing the Evolution of Chemical Process Simulation Models
96-13 *	K. Pohl / R. Klamma / K. Weidenhaupt / R. Dömges / P. Haumer / M. Jarke: A Framework for Process-Integrated Tools
96-14 *	R. Gallersdörfer / K. Klabunde / A. Stolz / M. Eßmajor: INDIA — Intelligent Networks as a Data Intensive Application, Final Project Report, June 1996

96-15 * H. Schimpe / M. Staudt: VAREX: An Environment for Validating and Refining Rule Bases
96-16 * M. Jarke / M. Gebhardt, S. Jacobs, H. Nissen: Conflict Analysis Across Heterogeneous Viewpoints: Formalization and Visualization
96-17 M. Jeusfeld / T. X. Bui: Decision Support Components on the Internet
96-18 M. Jeusfeld / M. Papazoglou: Information Brokering: Design, Search and Transformation
96-19 * P. Peters / M. Jarke: Simulating the impact of information flows in networked organizations
96-20 M. Jarke / P. Peters / M. Jeusfeld: Model-driven planning and design of cooperative information systems
96-21 * G. de Michelis / E. Dubois / M. Jarke / F. Matthes / J. Mylopoulos / K. Pohl / J. Schmidt / C. Woo / E. Yu: Cooperative information systems: a manifesto
96-22 * S. Jacobs / M. Gebhardt, S. Kethers, W. Rzasa: Filling HTML forms simultaneously: CoWeb architecture and functionality
96-23 * M. Gebhardt / S. Jacobs: Conflict Management in Design
97-01 Jahresbericht 1996
97-02 J. Faassen: Using full parallel Boltzmann Machines for Optimization
97-03 A. Winter / A. Schürr: Modules and Updatable Graph Views for PROgrammed Graph REwriting Systems
97-04 M. Mohnen / S. Tobies: Implementing Context Patterns in the Glasgow Haskell Compiler
97-05 * S. Gruner: Schemakorrespondenzaxiome unterstützen die paargrammatische Spezifikation inkrementeller Integrationswerkzeuge
97-06 M. Nicola / M. Jarke: Design and Evaluation of Wireless Health Care Information Systems in Developing Countries
97-07 P. Hofstedt: Taskparallele Skelette für irregulär strukturierte Probleme in deklarativen Sprachen
97-08 D. Blostein / A. Schürr: Computing with Graphs and Graph Rewriting
97-09 C.-A. Krapp / B. Westfechtel: Feedback Handling in Dynamic Task Nets
97-10 M. Nicola / M. Jarke: Integrating Replication and Communication in Performance Models of Distributed Databases
97-13 M. Mohnen: Optimising the Memory Management of Higher-Order Functional Programs
97-14 R. Baumann: Client/Server Distribution in a Structure-Oriented Database Management System
97-15 G. H. Botorog: High-Level Parallel Programming and the Efficient Implementation of Numerical Algorithms
98-01 * Jahresbericht 1997
98-02 S. Gruner/ M. Nagel / A. Schürr: Fine-grained and Structure-oriented Integration Tools are Needed for Product Development Processes
98-03 S. Gruner: Einige Anmerkungen zur graphgrammatischen Spezifikation von Integrationswerkzeugen nach Westfechtel, Janning, Lefering und Schürr
98-04 * O. Kubitz: Mobile Robots in Dynamic Environments

98-05 M. Leucker / St. Tobies: Truth — A Verification Platform for Distributed Systems

98-07 M. Arnold / M. Erdmann / M. Glinz / P. Haumer / R. Knoll / B. Paech / K. Pohl / J. Ryser / R. Studer / K. Weidenhaupt: Survey on the Scenario Use in Twelve Selected Industrial Projects

98-08 * H. Aust: Sprachverstehen und Dialogmodellierung in natürlichsprachlichen Informationssystemen

98-09 * Th. Lehmann: Geometrische Ausrichtung medizinischer Bilder am Beispiel intraoraler Radiographien

98-10 * M. Nicola / M. Jarke: Performance Modeling of Distributed and Replicated Databases

98-11 * A. Schleicher / B. Westfechtel / D. Jäger: Modeling Dynamic Software Processes in UML

98-12 * W. Appelt / M. Jarke: Interoperable Tools for Cooperation Support using the World Wide Web

98-13 K. Indermark: Semantik rekursiver Funktionsdefinitionen mit Striktheitsinformation

99-01 * Jahresbericht 1998

99-02 * F. Huch: Verifcation of Erlang Programs using Abstract Interpretation and Model Checking — Extended Version

99-03 * R. Gallersdörfer / M. Jarke / M. Nicola: The ADR Replication Manager

99-04 M. Alpuente / M. Hanus / S. Lucas / G. Vidal: Specialization of Functional Logic Programs Based on Needed Narrowing

99-07 Th. Wilke: CTL+ is exponentially more succinct than CTL

99-08 O. Matz: Dot-Depth and Monadic Quantifier Alternation over Pictures

2000-01 * Jahresbericht 1999

2000-02 Jens Vöge / Marcin Jurdziński: A Discrete Strategy Improvement Algorithm for Solving Parity Games

2000-04 Andreas Becks, Stefan Sklorz, Matthias Jarke: Exploring the Semantic Structure of Technical Document Collections: A Cooperative Systems Approach

2000-05 * Mareike Schoop: Cooperative Document Management

2000-06 * Mareike Schoop, Christoph Quix (Ed.): Proceedings of the Fifth International Workshop on the Language-Action Perspective on Communication Modelling

2000-07 * Markus Mohnen / Pieter Koopman (Eds.): Proceedings of the 12th International Workshop of Functional Languages

2000-08 Thomas Arts / Thomas Noll: Verifying Generic Erlang Client-Server Implementations

2001-01 * Jahresbericht 2000

2001-02 Benedikt Bollig / Martin Leucker: Deciding LTL over Mazurkiewicz Traces

2001-03 Thierry Cachat: The power of one-letter rational languages

2001-04 Benedikt Bollig / Martin Leucker / Michael Weber: Local Parallel Model Checking for the Alternation free μ-calculus

2001-05 Benedikt Bollig / Martin Leucker / Thomas Noll: Regular MSC languages

2001-06 Achim Blumensath: Prefix-Recognisable Graphs and Monadic Second-Order Logic

2001-07 Martin Grohe / Stefan Wöhrle: An Existential Locality Theorem

2001-08 Mareike Schoop / James Taylor (eds.): Proceedings of the Sixth International Workshop on the Language-Action Perspective on Communication Modelling

2001-09 Thomas Arts / Jürgen Giesl: A collection of examples for termination of term rewriting using dependency pairs

2001-10 Achim Blumensath: Axiomatising Tree-interpretable Structures

2001-11 Klaus Indermark and Thomas Noll (eds.): Kolloquium Programmiersprachen und Grundlagen der Programmierung

2002-01 * Jahresbericht 2001

2002-02 Jürgen Giesl / Aart Middeldorp: Transformation Techniques for Context-Sensitive Rewrite Systems

2002-03 Benedikt Bollig / Martin Leucker / Thomas Noll: Generalised Regular MSC Languages

2002-04 Jürgen Giesl / Aart Middeldorp: Innermost Termination of Context-Sensitive Rewriting

* These reports are only available as a printed version.

Please contact biblio@informatik.rwth-aachen.de to obtain copies.