Development and Configuration of Service-based Product Lines[*]

Alexander Gruler, Alexander Harhurin, Judith Hartmann
Technische Universität München
Department of Informatics
Chair of Software and Systems Engineering
Boltzmannstr. 3, 85748 Garching, Germany
{gruler,harhurin,hartmanj}@in.tum.de

Abstract

Increasing complexity due to the multitude of different functions and their interactions as well as a rising number of different product variants are just some of the challenges that must be faced during the development of multi-functional system families. Addressing this trend we present an approach combining model-based development with product line techniques aiming at a consistent description of a software product family as well as supporting the configuration of its variants. We integrate the concept of variability in our framework [7] which only supported the representation of single software systems on subsequent abstraction levels so far. For the configuration of a concrete product we extend this framework by a feature-based model which allows to configure and derive single systems from a system family model. Furthermore, we explain how the complexity due to the possibly huge amount of configuration decisions can be handled by means of a staged configuration process.

1 Introduction

Today, innovative functions – mainly realized by software – are one of the key potentials for competitive advantage in various application domains, e.g. the automotive domain. Increasing complexity due to a multitude of different functions and their extensive interaction as well as a rising number of different product variants are just some of the challenges that must be faced during the development of multi-functional system families.

Addressing this trend we present an approach combining model-based development along different abstraction layers with product line techniques aiming at a consistent description of a software product family as well as supporting the configuration of concrete variants.

The complexity stems from various sources, e.g. functional and architectural complexities or distribution. Thus, a promising approach for the development of multi-functional software systems is the use of different abstraction levels. In [7] we introduced a Layer Framework which supports the representation of software-intensive systems on subsequent abstraction levels. In this approach, a multi-functional system is modeled during the early phases of a model-based development process by means of a service model, the so-called *Service Diagram*. In the Service Diagram we use the concept of a *service* to independently model single functionalities of the system which are related and combined to form the overall system behavior. Thus, the Service Diagram captures the pure functionality of the system in a formal way and is the basis for further, more detailed architectural models, such as e.g. a logical component architecture. In particular, it establishes a formal relation between functional requirements and architecture models.

In this paper, we integrate the concept of variability in our Layer Framework and, thus, enlarge it to be applicable for the development of whole system families instead of single systems only. Thereby, we focus on the most abstract layer, the Service Diagram, and show how variability can be smoothly integrated in the existing service theory. Moreover, we extend our Layer Framework by an additional model, the *Configuration Model*. The Configuration Model is an integrated model combining the variability derived from functional as well as non-functional requirements. It comprises all variation points from all abstraction layers, which must be resolved to obtain a valid instance of the product line. Furthermore, we explain how the complexity due to the possibly huge amount of configuration decisions can be handled by means of a staged configuration process based on different views unto the Configuration Model.

[*]This work was partially funded by the German Federal Ministry of Education and Research (BMBF) in the framework of the VEIA project under grant 01ISF15A. The responsibility for this article lies with the authors.

Together with these enhancements our Layer Framework represents a promising approach combining model-based development of multi-functional systems on high abstraction layers with product line engineering techniques.

1.1 Contributions

In general, the software engineering process and respective methods for the development of complex, multi-functional systems have not reached a stage yet which satisfies the current needs of the industry. With the models and concepts described in this paper we address the following issues.

During the early phases of a model-based development process, i.e. during the transition from requirements to architecture models, an open issue is at what level to start best with a formal description. In practice today, functional requirements are not precisely formulated. The prevalent approaches to modeling requirements or the functionality offered by a system are use case diagrams [11] or – in the context of product line engineering – feature models [12] which both lack a precise semantics in general.

In contrast to a pure informal approach, we introduce with the Service Diagram a formal model with a well-defined semantics for describing the functionality (including variability) already in the requirements engineering phase. One of the advantages of such a formal model is that it allows an automatic analysis of the system. By this, discrepancies between conflicting functionalities can be detected and resolved already in an early phase of the development process.

In our approach, a system will be modeled at different levels of abstraction in a way that each level gives a more or less abstract view of the system. The fact that the models are based on the same notion of a service facilitates the transition from the Service Diagram to the subsequent layer (Logical Architecture). Thus, both models integrate seamlessly at top of such a model chain closing the formal gap in a model-based development process. In particular, they provide the basis for a formal transition from functional requirements to architecture design which currently is not well supported by formalisms.

Classical approaches to model-based development focus on the construction of a single product but do not explicitly address the relations between different configurable products nor support the description of variability. Thus, to be able to model a whole software product line, we enlarge our approach by concepts to represent variability and combine model-based development with product-line engineering which are generally considered in isolation.

While the majority of existing approaches only deal with modeling variability using informal techniques and mainly concentrate on structural aspects, we allow the modeling of behavioral variability in a formal way already in the early phase of the development process (Requirements Engineering). Thereby, we use the same notion of a service to model services of a single product as well as to model alternative services and variation points. Furthermore, as we formally integrate variability, automatic reasoning about product instances can be performed, e.g. proofing the existence of a valid instance, deriving all valid instances, etc..

Because of the possibly large amount of variability and the complex dependencies between different design artifacts, the correct derivation of single products is in general not a trivial task. Thus, we extend our Layer Framework by a new model, the *Configuration Model*, in order to support the challenging configuration process. On the one hand, all possible selection points, which must be resolved in order to obtain a valid instance of the software product line, as well as the dependencies between them, are explicitly represented in the Configuration Model. On the other hand, also the non-functional requirements, which heavily affect the configuration of a valid product, but which could not have been modeled in the Layer Framework before, are now incorporated in the Configuration Model.

Furthermore, we introduce the concept of a *view* partitioning the pool of all configuration decisions into suitable parts allowing each stakeholder to access only their relevant parts of the Configuration Model. Different views provide the basis for a staged configuration process which allows to eliminate variation points consecutively. This greatly helps to overcome the huge amount of configuration decisions and allows for leaving the end-user at the end of the "configuration chain", still having the maximum of freedom for customizing their product, while all choices, which are irrelevant from their point of view, have already been configured.

1.2 Running Example

The introduced techniques and models will be illustrated by the example of a door control unit (DCU) [10]. Since in a modern premium class car the whole functionality offered by the DCU is exclusively software based, it gives a realistic example of a multi-functional system with clearly distinguishable sub-functionalities.

The DCU provides a physical user interface consisting of several buttons in order to control several comfort features of a car. The functionalities are as expected, e.g. adjusting the seat in its horizontal and vertical axis, changing the angle of the seat back, saving the positions of the seat, turning on/off the seat heating and changing its degree of intensity, and more.

The various dependencies between different functions will be described at the appropriate places.

1.3 Outline

The rest of this paper is organized as follows: In Section 2 we recapitulate the existing Layer Framework. For the Service Diagram we introduce concepts to specify variability and dependencies. In Section 3 we describe the Configuration Model and different views unto this model. The resulting model-based development process for software product lines (Domain Engineering) based on the Layer Framework as well as the staged configuration process (Application Engineering) are presented in Section 4. Finally, we compare our models to related approaches in Section 5 before we conclude the paper in Section 6.

2 Model-based Development

This section gives a short introduction to our Layer Framework which consists of different abstraction layers. A more detailed description of this framework and its abstraction layers can be found in [7].

The basic idea of our approach is to specify a system on consecutive abstraction layers, each one giving a more detailed model of the system, where the highest layer reflects a very abstract, informal description of a system, while the lowest one represents a concrete deployable implementation. During the transition from a higher level to a more concrete one the system specification is enriched; i.e. the completeness and the precision-level of the design models are increased. Thereby, the transition from a higher layer to a lower one has to be proven correct: all specification constraints of the more abstract model have to hold in the more concrete one.

The Layer Framework (cf. Figure 1) starts from a very abstract description of the system as a set of informal requirements, e.g. use-cases without a well-defined semantics. The formalization of the informal functional requirements by independent services, their structuring, and their dependencies yield the next layer, called *Service Diagram*. Here, the system behavior is specified exclusively as being observable from the environment (black-box view). Refining the Service Diagram by adding communication behavior results in the consecutive model, the so-called *Logical Architecture*, which is a network of communicating services connected by channels. The next steps are to build Software/Hardware architectures and a deployable implementation. However, this is beyond the scope of this paper.

In the following, the basic concept of a *service* (the key component of our framework), and the Service Diagram which models the pure functionality of the system, are briefly recapitulated. Subsequently, the specification of hierarchical relations between modularly developed services of the *Service Diagram* is explained in detail. Lastly, we describe how the concept of variability is integrated into the existing service theory and show how variability dependencies are represented in the Service Diagram.

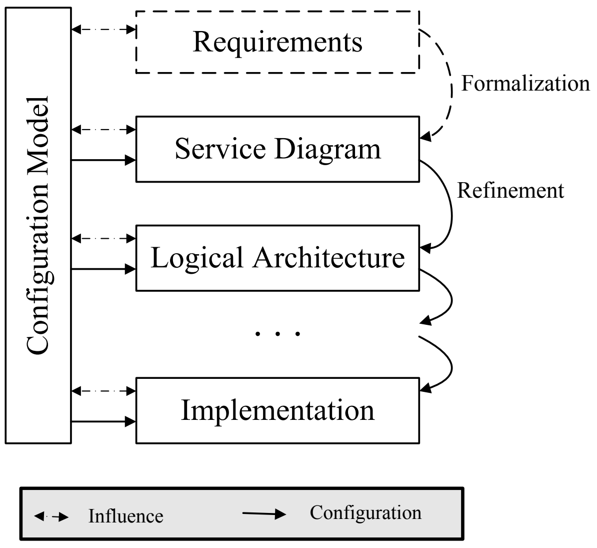

Figure 1. Layer Framework with Configuration Model

2.1 Service Theory

The central concept in our Layer Framework are services, which are used to modularly capture, structure, and relate the functionality offered by a system. The definition of a service is based on the idea of timed data streams as introduced in a Service Theory [2] by Broy.

Every service provides a syntactic interface and a behavioral semantics. The *syntactic interface* $I \blacktriangleright O$ of a service is given by the set of all its typed input ports I and output ports O. Given a service S with syntactic interface $I \blacktriangleright O$ for each port $p \in I \cup O$ and all time intervals $t \in \mathbb{N}$, the term $S[p](t)$ denotes the message communicated via the port p within the time interval t. The *behavioral semantics* of a service S is precisely characterized by a partially defined stream-processing function mapping streams of messages received on input ports $q \in I$ to streams of corresponding messages on output ports $p \in O$. Services can be connected through *channels* yielding whole systems.

The above sketch of the main principles of the Service Theory is rather short, but sufficient for our purposes. For more detailed information on services see [2].

2.2 Service Specification

In Section 2.1 services are formally defined by stream-processing functions. There are several ways of how to specify the behavior of a single service. One of them is an assumption/guarantee-specification as introduced in [3], which consists of two formulas, namely, an assumption and a guarantee. The assumption (a temporal logical predicate over inputs) specifies the domain of a service and the guarantee (a temporal logical predicate over inputs and outputs) characterizes the reaction of a service to its inputs, if and only if the inputs are in accordance with the assumption.

However, if a service is a set of reaction patterns which give a precise relation between inputs and outputs without any temporal properties, we propose to use modified I/O-automata [15] to specify a single service. An I/O-automaton constructively defines (infinite) input and output message streams as well as the relation between them.

All in all, a service is a suitable concept to modularly describe functionalities offered by a multi-functional, reactive system during early stages of the development where the focus of the developer is to model all functional information that is already known about the system.

2.3 Service Diagram

The last section showed how to specify an individual functionality of a system modularly and independently from other functionalities. Now we concentrate on the structuring of the functionalities of multi-functional systems and their dependencies. This results in a hierarchical structure of the system functionality where the overall functionality is decomposed in services and sub-services, with defined relations between them.

The *Service Diagram* gives a specification of the behavior of a system as observable from the environment when viewing the system as a black-box. Both, the individual services offered by a system and the dependencies between them are specified, but we do not consider the architecture of the system (i.e. communication links between services). Thus, we formally specify functional requirements without any prognosis about the implementation. In particular this implies that we consider the whole system as a single (but complex) service itself composed of several sub-services.

The Service Diagram consists of a set of services and four kinds of relationships between them, namely *refinement*, *aggregation*, *variability*, and *dependency* (cp. Figure 2). The relationships refinement, aggregation and dependency are already introduced in detail in [7] and will be only sketched here. The focus of this section lies on dependencies arising from variability since these are essential for the specification of families of related products.

Refinement Since a typical multi-functional system offers a plethora of functions with complex interactions between them, representing all this information without abstraction would have a negative effect on the usability of the specification. To master the complexity of a specification, we introduce *abstract services*.

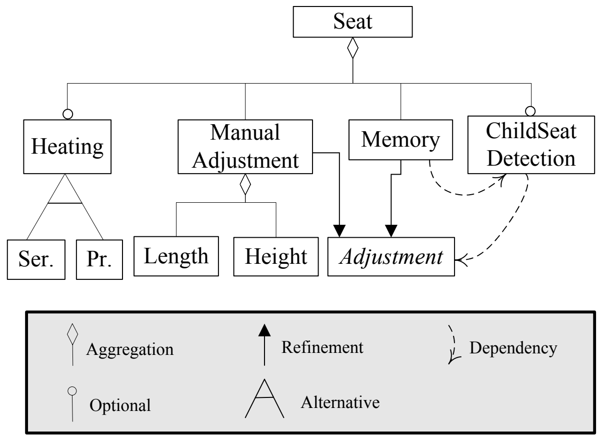

Figure 2. Service Diagram

An abstract service is an abstract specification of one or several functionalities. It can be considered as a contract between the services refining it and the environment – when a service refines an abstract service, it promises to provide at least the behavior already specified by the abstract service. An abstract service cannot be implemented directly, but rather must be refined by other services – it only helps to structure services and dependencies between them.

Aggregation The aggregation relations allow to arrange individual services which have been specified independently into a service hierarchy. Thus, it greatly helps to reduce the complexity of the system functionality.

Aggregation is defined as a relation between a service and its sub-services. It directly reflects the idea that the functionality offered by a service can be subdivided into different sub-functionalities. A sub-service specifies a subfunctionality of its super-service, in contrast to a refining service which refines the functionality of a *whole* service.

A super-service composed of several sub-services is called a *compound service*. According to the aggregation relation we define the semantics of a compound service as a container of all *concurrently* operating sub-services. The behavior of a compound service can be reproduced from its sub-services using the well-defined semantics of the aggregation relation. Some of the sub-services of a compound service may influence each other. Then, additionally to the aggregation, their mutual dependencies must be defined, because a sub-service may be restricted by other sub-services when being combined within a parent compound service.

Variability The Service Diagram as introduced so far can only specify a single software system.i.e. where all specified services will actually be present in the final system. With the new concept of variability, we enlarge our approach by product line techniques aiming at the explicit modeling of commonalities and variabilities of different, though similar, services. To explain our approach in the scope of this paper we use only the two types of variability (optional and alternative) introduced by Kang et al. in [12]. However, it is possible to enlarge it by other variability relations like [6] introduced by Czarnecki et al..

The definition of the *alternative relation* is similar to those of the aggregation. It is defined as a relation between a service (*variation point*) and a set of mutually alternative sub-services (*variants*). Such a variation point is defined for all inputs for which one of its alternative sub-services yielded a defined output as well. In contrast to a compound service whose semantics is defined as a container of *all* concurrently operating sub-services, the functionality of an alternative variation point is specified by *exactly one* of its sub-services – its behavior can be reproduced from one of its alternative sub-services. This means that there is an implicit *excludes*-dependency between all sub-services that belong to the same variation point (see the following paragraph for further details about dependencies). In our example, *Heating* is an alternative variation point which is specified either by its sub-service *Ser.* (serial heating) or its sub-service *Pr.* (premium heating).

Optional is the second variability relation in our framework. It is a relation between a service (a variation point that might consist of some mandatory sub-services) and an optionally selectable sub-service. Since the optional relation is merely an abbreviation of the alternative one, the optional variation point is considered as an alternative one which consists of two variants: the optional variation point with and without the selectable sub-service. In our example, *Seat* is an optional variation point which comprises two optional sub-services (*Heating*, *Child Seat Detection*) as well as two mandatory ones (*Memory*, *Manual Adjustment*).

Dependency By dependencies, we mean relations between services in a way that the operation of one service depends on those of other services. Thus, a dependency does not give structural information but describes how services might effect each other. Although there are a lot of methodological significant dependencies like *disables* or *influences*, the aim of this paper is to present techniques to specify dependencies rather than to completely enumerate all of them. The following two dependencies – which are essential for the modeling of a product line – should be noted. Two services might *exclude* each other, i.e. there is no system in the product line which offers both functionalities specified by the two services respectively. A service might *require* another one, i.e. there is no system in the product line which offers the functionality specified by the former service without offering the functionality specified by the latter one.

Dependencies between services are specified as being observable from the overall system boundaries without changing their interfaces (no additional ports are added) and without characterizing the communication between them (no additional channels are added). To specify these dependencies we introduce additional constraints. A *constraint* restricts the behavior of the influenced service by defining dependencies between its I/O message streams and those of the influencing service. These constraints are defined by first order logic expressions over names of services, ports as well as access operations and specify dependencies between port values of different services in time intervals. In our example, there is an implicit *excludes*-dependency between two alternative services *Ser.* (S) and *Pr.* (P). Since these services are alternative, a system can offer either a serial or a premium heating functionality. This dependency for example is specified by the following constraint:

$$(\exists t \in \mathbb{N} : \exists p \in I_S \cup O_S : S[p](t) \neq \epsilon) \Rightarrow$$
$$(\forall t' \in \mathbb{N} : \forall p' \in I_P \cup O_P : P[p'](t') = \epsilon)$$

This means, if there is a point in time in which the service *Ser.* receives or sends a message on its input I_S or its output ports O_S respectively (i.e. the service is present in the system) then in all points in time the service *Pr.* receives or sends no message on its input I_P or output ports O_P respectively (i.e. the service is absent in the system).

Note, that the modular specifications of individual services are not modified when defining constraints. Constrains only specify the interplay between services which must be satisfied by the components of the Logical Architecture. In other words, constraints provide criteria to verify the models of the consecutive layer. Only those models which do not violate these constraints are valid.

Altogether, the Service Diagram allows to model system functionalities independently by simple services, arrange these services into a service hierarchy, explicitly model commonalities and variabilities, and specify dependencies between these individual services. Thus, it establishes the foundation for the behavior specification of an entire product family.

3 Configuration Model

In the previous section we have extended the Layer Framework by the ability to model whole families of related software systems rather than only a single software system.

Because of the possibly large amount of variability and the complex dependencies between different design artifacts, the correct derivation (*configuration*) of single product instances is not a trivial task. Thus, in order to support

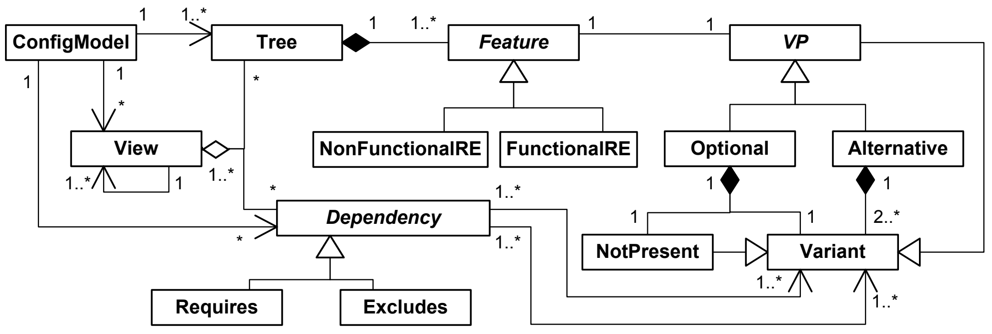

Figure 3. Configuration Meta-Model

the configuration process, we introduce a separate *Configuration Model* (cp. Figure 1). This model represents all possible selection points, which must be resolved in order to obtain a valid instance of the software product line.

The Configuration Model comprises a set of variation points and their variants respectively, which can be selected by different stakeholders, e.g. customers, component suppliers or designers during different stages of the development process. Furthermore, dependencies between selected variants are made explicit here. The variability and the dependencies reflect the information already incorporated in the design models. With *design models* we mean the models developed according to the Layer Framework, e.g. the Service Diagram or the Logical Architecture. In addition, non-functional requirements – not yet regarded in the design models – are also incorporated within the Configuration Model.

Depending on the stakeholder as well as the phase of the development process, we offer different views on the Configuration Model presenting each stakeholder only those selection points which are relevant for them. In particular, this facilitates a staged configuration process.

In the following, we introduce structure and technical aspects of the Configuration Model, relate it to informal requirements and the artifacts of the design models, and explain usage and realization of different views on the Configuration Model. The Configuration meta-model is shown in Figure 3 using a class diagram formalization.

3.1 Structure of the Configuration Model

We assume that a product line is fully described by a set of features. A *feature* is "a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems" [12]. In the Configuration Model, we only consider features which have different variants and call them *variation points*. The other (so-called mandatory) features can not be configured by the user and therefore are not shown in this model.

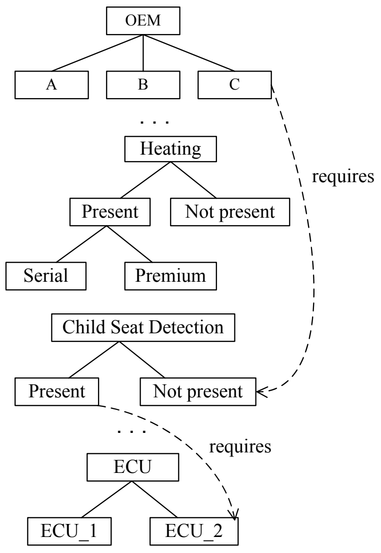

Figure 4. Configuration Model

The Configuration Model consists of a set of variation points $F = \{F_i \mid i = 1..n\}$, each point F_i has a set of variants $F_i = \{V_{i,j} \mid j = 1..k\}$. Each variant of a feature can be a variation point itself, thus, we allow to build-up hierarchical feature trees.

In general, each variation point of a design model results in a corresponding feature within the Configuration Model. For the Service Diagram, each optional service leads to a feature with two variants (present or not), and each variation point together with its alternative sub-services directly results in an equivalent feature and its variants within the Configuration Model. The same applies analogously to the design models of the consecutive abstraction layers (Logical Architecture, Implementation etc.). There is a 1:1 mapping between these features and variation points of the design models in order to support the configuration of end-products solely based on the Configuration Model. Further features are derived from the non-functional requirements which are not considered in the design models. This non-functional information is only used to select artifacts from alternative ones with the same functional features. Therefore, these non-functional features have no mapping to the design models.

In our example, the variability of the Service Diagram (cp. Figure 2) is expressed in two features: *Heating* and *Child Seat Detection* (cp. Figure 4). While the optional service *Child Seat Detection* accounts for the feature *Child Seat Detection* with only two variants (present or not), the service *Heating* results in a hierarchical feature *Heating*. During a configuration process one has to decide on the existence (present or not) and on the type of the seat heating (serial or premium). The variation point *ECU* originates from the hardware layer, whereas the different choices for the *OEM* are derived solely from the non-functional requirements.

Moreover, the Configuration Model comprises a set of dependencies $D = R \cup E$ between feature variants (in this paper we only consider requires and excludes dependencies, R and E respectively). Formally, D is a relation over the set of all possible permutations of feature variants: $D \subseteq \times_{i=1\ldots n} F_i$. The dependencies restrict the set of valid products that can be configured and guide the configuration process (see Section 4). They originate from the following causes: Firstly, they reflect dependencies specified within the design models. Such dependencies in the Service Diagram are introduced in Section 2.3. Analogously, the dependencies specified on the other abstraction layers must be taken into account. Secondly, as the Configuration Model is an integrated model combining the variability of the design models of all abstraction layers, it allows for explicitly specifying dependencies between the design artifacts of consecutive layers. Thirdly, the dependencies are used to reflect non-functional requirements, e.g. marketing decisions.

In our example, the requires dependency between variants of *Child Seat Detection* and *ECU* describes the (hypothetical) fact that the existence of a child seat detection necessarily causes the complex ECU-solution *ECU_2* consisting of two separate ECUs for the driver and the co-driver seat. Thus, it describes the relation of the Service Diagram to the Hardware Architecture. The second relation (between variants of *OEM* and *Child Seat Detection*) reflects the (hypothetical) informal requirement that a car of *OEM C* must never have a child seat detection.

Thus, the Configuration Model is formally specified by the pair $C = (F, D)$, which semantics is defined as $[\![C]\!] \subseteq \times_{i=1\ldots n} F_i \setminus NV$, where $NV = \{p \in \times_{i=1\ldots n} F_i \mid p \in E \vee p \notin R\}$ is a set of non-valid products, i.e. that violate a dependency of D.

3.2 Relation between Features and Informal Requirements

In Section 3.1 we introduced features as basic entities of our Configuration Model. However, the set of features is not equivalent to the set of informal requirements, as not all requirements are expressable as features. The informal requirements can be divided into three different categories:

- **high-level requirements**: business requirements dealing with the question of how the company might profit from the system to be developed or other high-level requirements referring to legal restrictions and more (e.g. “The product must be adapted for the Asian market”).
- **process requirements**: requirements which restrict only the development process but not directly the product (e.g. “The development process shall be conforming to the rational unified process”).
- **product requirements**: all functional requirements defining the functionality the system should offer its users, and non-functional requirements constraining the software and hardware architectures of the system.

Features, as used in our approach, cover exactly the product requirements, while business and process requirements are disregarded. Even though, we only consider software-based systems, we do not force a feature to be directly software-based itself, e.g. in the automotive domain the property of having leather upholstery is also considered to be a regular feature of a system. But we only allow features which influence (directly or indirectly) the resulting software of the system.

3.3 Different Views unto a Configuration Model

The main purpose of the Configuration Model was to capture *all* features that require a selection of one of its variants in the following configuration process. This means, that the Configuration Model does not differentiate between features that are configured by different stakeholders or at different points of time during the staged configuration process. From this point of view it is simply a general repository of all pending feature configuration choices.

Certainly, for a development process which involves many parties (e.g. OEM, supplier, end-user) and which is split up into various development stages (functional design, architecture development, system integration), such a model becomes quickly too complex and not directly usable anymore. Therefore, we allow to define different views unto the Configuration Model.

Formally, a *view* W is defined as a pair (F_W, D_W), where $F_W \subseteq F$ and $D_W \subseteq D$. This means, a view consists of a subset of features and a subset of inter-feature dependencies that are already defined in the Configuration Model. Additionally, not all variants of a feature must necessarily be present in a view, i.e. $F_{i_W} \subseteq F_i$ for $i = 1..n$.

Views offer several advantages. Firstly, with the concept of views we allow to reduce the complexity of the entire Configuration Model by showing only a relevant extract. Secondly, by analyzing the existing dependency chains between features, a view allows to define a causal order on how the selection process has to be performed within a single view. Thus, the concept of views facilitates a staged configuration process by defining a way to proceed through the feature configuration choices of every single view.

In our example, a view individually defined for the OEM *C* (i.e. the variation point *OEM* is resolved by its variant *C*) consists only of two features *Heating* and *ECU*, since due to the dependency between *C* and *Not Present* of *Child Seat Detection* the latter is automatically resolved by *Not Present* (cp. Figure 4).

4 Process

Having introduced our Layer Framework, we briefly sketch the resulting model-based development process for software product lines in this section. Basically, the process can be divided into two parts: the *Domain Engineering* process, which deals with the construction of the design models and the identification of all selection points, and the *Application Engineering* process, which prepares the configuration model so that the product instances can be configured only by high-level selection points visible for the end-user.

Domain Engineering The development of a product line is guided by the abstraction layers (cp. Figure 1). Starting point of the software development process typically is a (more or less) complete collection of functional and non-functional requirements being documented in an arbitrary way. The requirements are not or only loosely related to each other, i.e. it is difficult to identify unwanted interactions or contradictory requirements. In the Service Diagram, the system behavior is being specified from the point of view of the the environment – the functional requirements are modularly described by services which are directly observable by the environment. Subsequently, observable inter-service dependencies are specified in order to detect contradictions and interactions. Refining the Service Diagram by connecting the identified services by channels and adding service intercommunication behavior yields the Logical Architecture. Thereby, the refinement must be proven correct, i.e. in spite of the integration of additional information the Logical Architecture must comply with the information incorporated in the Service Diagram. On the subsequent layers, the model information is further enriched step by step resulting in a complete specification of the overall system. However, the process is not intended to be performed strictly top-down but rather in a iterative way.

Mutually influencing each other, the features of the Configuration Model are developed concurrently to the artifacts of the abstraction layers. On the one hand, the variation points of the design models are directly reflected in the Configuration Model (see Section 3.1). On the other hand, it is also possible and sometimes very helpful to start with the identification and informal description of (part of) the variability within the Configuration Model first and design the corresponding solutions within the respective design model afterwards. In any case, with each step the set of possible choices (the variation points of the Configuration Model) increases, since a concrete design artifact of an abstract layer can possibly be implemented by several different solutions in a less abstract layer. In the end, the Configuration Model represents the pool of all pending configuration decisions.

Application Engineering To obtain a valid system, a suitable variant of each of the numerous variation points of the Configuration Model must be selected. To overcome the huge amount of configuration decisions, we propose a staged configuration process consecutively eliminating variation points. Bottom-up, starting with the most concrete layer, the corresponding variation points are being resolved by generally selecting one of the variants or by relating the variants (by excludes/requires dependencies) to more abstract features. Thereby, the non-functional requirements are being taken into account. Consequently, these variation points must not be considered anymore on the more abstract layers. The same applies for the remaining layers until a completely configured product can be derived out of the product family specification.

This corresponds to a staged configuration process, where each stakeholder can only make their selections independently from the previous or remaining selections. For this purpose, consecutive views (see Section 3.3) are an adequate means to reflect a causal order of the configuration process, i.e. the configuration according to one view should precede the configuration of another view. The idea is that a single view reflects all configuration choices (configurable features) that a certain stakeholder (for whom the view is intended) can perform in a certain development stage. After that, the product is configured as far as possible, i.e. as many variation points as possible are resolved. Only those configuration decisions will be passed to the next stakeholder involved in the configuration process, which could (or should) not have been resolved in the current view. Finally, all choices that are not directly visible for the end-user are eliminated or appropriately related to the variants to be selected by the end-user.

This leaves the end-users at the end of the "configuration chain", ensuring that they still have the maximum of freedom for customizing their product, while all (irrelevant implementation) choices, e.g. which are not relevant from their point of view, have already been configured.

5 Related Work

The approach presented in this paper introduces variability into the Layer Framework [7] consisting of hierarchically structured system specifications. Thus, related approaches can be mainly found in two different areas: model-based and product line development.

Model-Based Development In model-based development, every important aspect of a software system is described by models. An important work in this area is the generative software development [4] introduced by Czarnecki. This approach focuses on automating the creation of systems: a system can be automatically generated from a textual or graphical specification. However, this approach as well as approaches like [18, 20] focus on the construction of a specific solution on a more detailed level of abstraction (e.g. software architecture) without supporting the formal requirement specification.

In contrast, we concentrate on the formalization of functional requirements and close the formal gap between requirements and architecture design in the early phase of the model-based development process. This idea is not new – the structuring of requirements and the relationship between requirements and architectures have received attention in recent years. The goal-based approach introduced by Lamsweerde [21] performs a transition from functional and non-functional requirements to architecture. The software architecture is built based on functional requirements derived from system goals, and subsequently refined by means of non-functional requirements. Medvidovic et al. [17] proposed an approach intended to provide a systematic way of reconciling requirements and architectures using intermediate models. However, these approaches as well as approaches like [9, 22] specify requirements and architectures by means of different models, and the model transformation process has several formal gaps. In contrast, by using services to structure requirements we are able to identify components for a prospective architecture already during the requirements engineering phase. The formal relationships between features, services and components allow a dynamic adjustment of the architectural structure depending on chosen requirements, which is a prerequisite for the product line development.

The closest approach to our work is a theoretical framework introduced by Broy [2] where the notion of a service behavior is formally defined. This framework provides several techniques to specify and to combine functionalities based on their behaviors. However, this quite theoretical approach does not cover several relevant issues what our work focuses on (e.g. formalization of variability or techniques for the specification of inter-service relations).

Product Lines Development A large number of contributions have been made over the past decade in order to specify software product lines. Our work was significantly inspired by the FORM process [13], and particularly by its practice of layering of design models. However, classical approaches to feature-oriented development like [12, 8, 14] only focus on the modeling of relationships between features, using uninterpreted features as the corresponding basic concept. The second deficit of these methods is that the absence of a formal semantics of the graphical notations prevents an automatic analysis of them. In contrast, in our approach the behavior of single features as well as the semantics of their relationships are specified.

The approach of formalizing feature models is not new. Several authors [1, 16, 6] define a formal semantics for the feature modeling language by means of different kinds of logics or grammars. As mentioned above, the main deficit of these approaches is disregard for the behavior of single features. "As a consequence, these approaches focus on the analysis of dependencies, however abstracting away from the causes for these dependencies" [20].

The value of behavioral information associated with individual features has also been recognized by others. In [5], Czarnecki and Antkiewicz propose an approach to mapping feature models to other models, such as behavior or data specifications, in order to give them semantics. However, this approach only focuses on assets like software components and architectures and not on the formalization of user requirements and their analysis in the early phases of the development process.

An approach to defining an orthogonal variability model was introduced by Pohl et al. in [19]. This model relates the variability to other software development models such as use case or component models. However, this approach does not integrate the variability into the development models. In contrast, we concentrate on the formalization of behavior variability within the design models and use the Configuration Model only to support the configuration process.

Our Configuration Process was inspired by the Staged Configuration introduced by Czarnecki et al. [6] – a transformation process that takes a feature diagram and yields another one where each stage eliminates some configuration choices. The drawback of the use of feature diagrams without behavior is already explained above.

6 Conclusion

The presented concepts can be roughly summarized as follows: Based on an existing framework for a layered model-based development process, we have introduced concepts to handle the development of whole software system families instead of single systems only. On the one hand we have extended the existing design models by concepts to model variability, on the other hand we have introduced an additional model – the Configuration Model – together with the concept of different views unto it, in order to support the challenging configuration process. Finally, we have sketched the development and configuration processes for a family of software systems based on the enhanced Layer Framework. We illustrated our concepts with an example from the automotive domain.

With the Service Diagram we provide a formal model with a well-defined semantics for describing the functionality already at an early stage of the development process. Due to the integration of variability, the Service Diagram now allows to formally capture the (functional) requirements of an entire product family. Thus, discrepancies between conflicting functionalities can be detected and already resolved in the requirements engineering phase of the development process. Besides, this permits automatic analysis of product lines. Moreover, since the Service Diagram seamlessly integrates at top of the layer framework, it helps to bridge the gap between usually informally specified requirements and formal design models.

The Configuration Model together with the concept of views allows us to capture another group of requirements: Non-functional requirements (NFRs) which are related with the configuration process, and which could not have been modeled in the Layer Framework before. Such kinds of NFRs can now directly be reflected in the set of variants and dependencies that are presented in the Configuration Model as well as in the structure of different views.

Altogether, by incorporating variability concepts in our Layer Framework we have achieved to widen our model-based development approach in the application area of product-line engineering. Concerning the Application Engineering, the Configuration Model and the concept of different views on it result in a staged configuration process which is suitable to deal with the derivation of single systems from highly complex product-line models.

References

[1] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Trans. Softw. Eng. Methodol.*, 1(4):355–398, 1992.

[2] M. Broy. Service-oriented systems engineering: Modeling services and layered architectures. In *FORTE*, pages 48–61, 2003.

[3] M. Broy and K. Stoelen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer, 2001. ISBN 0-387-95073-7.

[4] K. Czarnecki. Generative software development. In *SPLC*, page 321, 2004.

[5] K. Czarnecki and M. Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In *GPCE*, pages 422–437, 2005.

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

[7] A. Gruler, A. Harhurin, and J. Hartmann. Modeling the functionality of multi-functional software systems. In *Proceedings of 14th Annual IEEE International Conference on the Engineering of Computer Based Systems (ECBS)*, March 2007.

[8] J. V. Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. *wicsa*, 00:45, 2001.

[9] J. G. Hall, M. Jackson, R. C. Laney, B. Nuseibeh, and L. Rapanotti. Relating software requirements and architectures using problem frames. In *RE '02: Proceedings of the 10th Anniversary IEEE Joint International Conference on Requirements Engineering*, pages 137–144, 2002.

[10] F. Houdek and B. Paech. Das Türsteuergerät – eine Beispielspezifikation. http://www4.in.tum.de/lehre/vorlesungen/ase/ss05/iese-002_02.pdf (in German), accessed 14.02.2007.

[11] I. Jacobson. Use cases and aspects - working seamlessly together. *Journal for Object Technology*, 2(4):7–28, July/August 2003.

[12] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, SEI, CMU, Pittsburgh, PA, 1990.

[13] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. Form: A feature-oriented reuse method with domain-specific reference architectures. *Ann. Softw. Eng.*, 5:143–168, 1998.

[14] K. C. Kang, J. Lee, and P. Donohoe. Feature-oriented project line engineering. *IEEE Softw.*, 19(4):58–65, 2002.

[15] N. A. Lynch and M. R. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, 1989.

[16] M. Mannion. Using first-order logic for product line model validation. In *SPLC*, pages 176–187, 2002.

[17] N. Medvidovic, P. Gruenbacher, A. Egyed, and B. W. Boehm. Bridging models across the software lifecycle. *J. Syst. Softw.*, 68(3):199–215, 2003.

[18] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *SIGSOFT Softw. Eng. Notes*, 17(4):40–52, 1992.

[19] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering*. Springer, 2005.

[20] B. Schätz. Combining product lines and model-based development. In *Proceedings of Formal Aspects of Component Systems (FACS 2006)*, 2006.

[21] A. van Lamsweerde. From system goals to software architecture. In *SFM*, pages 25–43, 2003.

[22] W. Zhang, H. Mei, H. Zhao, and J. Yang. Transformation from cim to pim: A feature-oriented component-based approach. In *MoDELS*, pages 248–263, 2005.