Modeling the Functionality of Multi-Functional Software Systems*

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

Today, many software-based, reactive systems offer a multitude of functionality. One way to master the development of such a system is to model its functionality on an abstract level and derive a system architecture and an implementation out of this functionality model. In this paper, we present an approach to model the functionality by means of related, interacting services. For us, a service represents an single functionality of a system. The concept of services is used in two consecutive model layers with well-defined semantics leading from a black-box description of the system to a white-box model which consists of communicating services. Due to the precise semantics of a service and the interaction of services, the service models can be directly refined to a logical component architecture which in turn integrates into the development of a concrete implementation of the overall system.

1 Introduction

Today, many software-intensive systems provide a wide range of functionality, i.e. they offer a variety of different, user-observable functions. We call such systems *multi-functional*. For us, a *multi-functional system* is any reactive system which offers a set of different functions and combines them into a single system in consideration of their mutual dependencies. Thereby, the functionality of the overall system exhibits a surplus value compared to the set of individual functions. With *functionality* we mean the characteristic, observable behavior of a system, or more precisely its reaction (outputs) to certain inputs.

*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.

The property of multi-functionality spans various application domains such as telecommunication, avionics or automotive. Here for example, the functionality offered by an automobile has been increasing rapidly during the last decades, having reached a state where a modern premium class automobile has advanced to be a highly versatile multi-functional system. Today, the system functionality is mainly realized by software. Certainly, the efficient development of software for such complex, multi-functional systems requires special techniques and methods.

Addressing this trend, we introduce an approach to model a multi-functional system during the early phases of a model-based development process by means of two integrated *service models*. The service models capture the pure functionality of the system in a formal way and are the basis for further, more detailed architecture models, such as e.g. a logical component architecture. In particular, they establish a formal relation between functional requirements and architecture models. In both service models the concept of a *service* is used to independently model single functionalities of the system which are related and combined to form the overall system behavior.

We call the upper, more abstract model the *Service Diagram*. It gives a structured view of the overall functionality offered by the system as a hierarchy of all services directly observable by the user/environment. Subsequently, the Service Diagram is refined into a consecutive, less abstract model, the so-called *Service Network*. It gives a more detailed view on the system, by now considering the interaction between the identified services. Together with the Service Diagram, the Service Network provides the basis for the construction of the *Logical Architecture*.

The rest of this paper is organized as follows: In Section 2 we briefly introduce an example of a multi-functional system, which will be used throughout the rest of the paper to illustrate the suggested concepts. Section 3 presents current issues in the modeling of multi-functional systems and motivates the presented techniques. In Section 4 we introduce

and briefly define the basic concepts such as the idea of a *service* and motivate their relevance for the use as means of describing the functionality of multi-functional systems. Section 5 can be seen as the core of the paper, since here, we describe how the basic concepts should be applied in order to formally capture the functionality of a system. In particular, we describe the two service models as well as the model of the logical components. Finally, we compare our models with related approaches in Section 6 before we conclude the paper in Section 7 together with an outline how to integrate the concepts in our future work.

2 Running Example

The introduced techniques and models will be illustrated by the example of a door control unit (DCU) [13]. 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 clear distinguishable sub-functionalities. In the following, we briefly describe the functionality of the DCU.

The DCU controls several comfort features of a car, such as adjustment of the power front seats, memory functionality, seat heating, door lock, power windows, interior lighting, and the adjustment of the outside mirrors.

The DCU provides a physical user interface consisting of several buttons and switches mainly located in the front and back door lining. The functionalities are as expected, e.g. adjusting the seat in its horizontal and vertical axis, changing the angle of the seat back and the extension of the head restraint, opening and closing the windows, saving the positions of the seat, mirrors and steering wheel, turning on the seat heating and changing its degree of intensity, and adjusting the vertical and horizontal angle of the outside mirrors and turning on/off the mirror heating.

Dependencies between different functions and other relevant details will be described at the appropriate places.

3 Contributions

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

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 with a formal description. In practice today, functional requirements are not precisely formulated. The usual approaches to modeling requirements or the functionality offered by a system are use case diagrams [14] or feature models [15] which both lack a precise semantics [8] in general.

In contrast to a pure informal approach, we introduce a formal model with a well-defined semantics for describing the functionality already at an early stage of the development process. This has several advantages: Firstly, a formal model which formalizes (functional) requirements allows an automatic analysis of the system already in the early phases of the development process. By this, discrepancies between conflicting functionalities can be detected and resolved. Secondly, a formal model can be simulated – in the case of the service models with the CASE-tool AutoFocus [21] – which is a valuable property for industrial application.

The increasing complexity of multi-functional systems requires to design a system in a modular fashion by splitting up the system into an appropriate set of different sub-functions and generating the overall system as a combination of these. This implies to model sub-functionalities independently and to compose/combine them adequately afterwards in order to form the overall system behavior. In both our models we realize a modular, independent specification of (sub-) functions by means of modular *services*. A single service can consist of several sub-services with the meaning that a service aggregates the functionalities expressed by its sub-services. On the one hand, this allows a great freedom in the specification of an individual service, but on the other, it results in a structured hierarchy of services which forms a single service representing the overall functionality of the system.

Concerning the development process 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 both models are based on the same notion of a service facilitates the transition from the Service Diagram to the Service Network. Thus, both service models integrate seamlessly at the 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.

The abstraction level where a system is seen only according to its functionality is an appropriate place to realize changes due to an evolution of the system. Every functionality modularly modeled here can be traced to a set of implementation entities (such as components) in subsequent models, which means that changes in the service models can easily be propagated in subsequent models. Thus, the service models represent a suitable abstraction level with a high re-use potential.

4 Service Theory

Before we describe the service layers in detail in Section 5 we introduce the necessary basics in this section. The following definitions and ideas are based on the Service Theory [3] introduced by Broy which itself is based on the FOCUS theory [5] for the specification of interactive systems. Since the FOCUS theory assigns a precise semantics to each of its concepts, it is a suitable basis for the ideas introduced in this paper.

We use the concept of services to capture, structure and relate the functionality offered by a system. So far, this is similar to *feature*-based approaches (see Section 6), but in contrast to a feature a service has a precise semantics defined by its input/output behavior.

Basically, a service is based on the idea of timed data streams which are used to model the interaction of a service by describing the communication with its environment. Intuitively, a *timed (data) stream* can be thought of as a chronologically ordered (finite or infinite) sequence of data messages. Given a set M of data messages, we denote a timed stream of elements from M by a function $s : \mathbb{N} \rightarrow M$. We assume a model of time consisting of an infinite sequence of time intervals of equal length. In each time interval only one message can be transmitted. Such streams can be used to represent histories of communications of data messages transmitted within a time frame. For each time interval $t \in \mathbb{N}$, $s(t)$ denotes the message communicated within the time interval t.

Every service provides a syntactic service interface and a behavioral semantics. The *syntactic interface* of a service is given by the set of all typed ports of the service. We write $I \blacktriangleright O$ to denote the service interface, where I is the set of input ports and O the set of output ports respectively. With every port we associate a stream representing the message history of this port. 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. Note that a service can interact with its environment exclusively via its ports.

The *behavioral semantics* of a service S with syntactic interface $I \blacktriangleright O$ is precisely characterized by a partially defined (stream-processing) function mapping streams of messages received on the input ports $q \in I$ to streams of corresponding messages on the output ports $p \in O$. With "partially defined" we mean that a service does not always have to return a well-defined output, i.e. the stream-processing function, which characterizes the behavior of the service, does not have to be defined for all possible inputs. With $Dom(S)$ we denote the set of input streams of the service S which have defined outputs.

Services can be connected through *channels*. The idea is that a directed channel (p, q) between two ports p and q represents a connection between two services S_1 and S_2 with compatible[1] service interfaces $I_1 \blacktriangleright O_1$ and $I_2 \blacktriangleright O_2$, where $p \in O_1, q \in I_2$ respectively.

All in all, a service is an appropriate concept to 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 information that is already known about the system while not bothering about the situations which are not important for the current level of abstraction. In the next section we will see how these concepts can be used in order to describe a system as a set of related services.

5 Functionality Model Layers

In this section we introduce a framework of layers which gives different views on the system from different levels of abstraction respectively (see Figure 1). This corresponds to the idea 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 the systems, while the lowest layer represents a concrete deployable implementation.

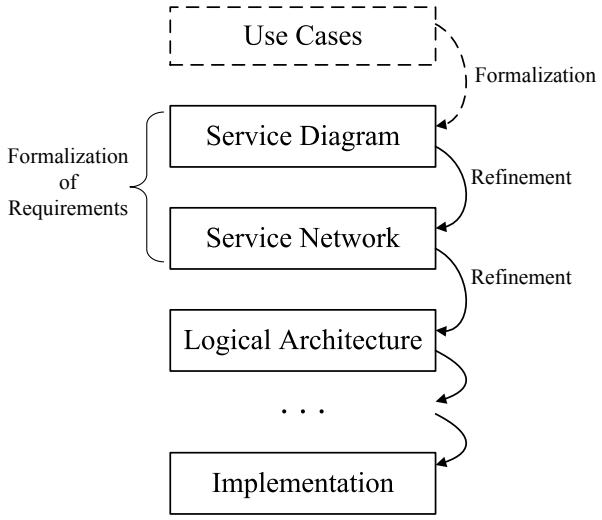


Figure 1. Layer Framework

Our layer framework starts from a very abstract description of the system as a set of use-cases or feature models without a well-defined semantics.

The formalization of use cases or features by independent services and their structuring yield the next layer, called *Service Diagram*. In this model, the system behavior is specified from the environment point of view (black-box

[1] For the complete definition of a connection between services see [3].

view). Therefore, each functionality is described by a service which is directly observable by the environment, i.e. its inputs can directly be triggered from the environment and its outputs are directly observable from the environment. Subsequently, observable inter-service dependencies are specified. Note that here, we do not characterize the communication between individual services – we only specify dependencies between them as being observable from its overall system boundaries.

Refining the Service Diagram by adding communication behavior yields the consecutive layer, called *Service Network*. It gives a more detailed view on the system, by now considering the interaction of the identified services and focusing on their intercommunication. This results in a network of communicating services which realize the functionality modeled in the Service Diagram.

The last step of our approach is to build up a *Logical Architecture* formed by a network of components, which are connected by channels, and to refine the functionality specified in the Service Network.

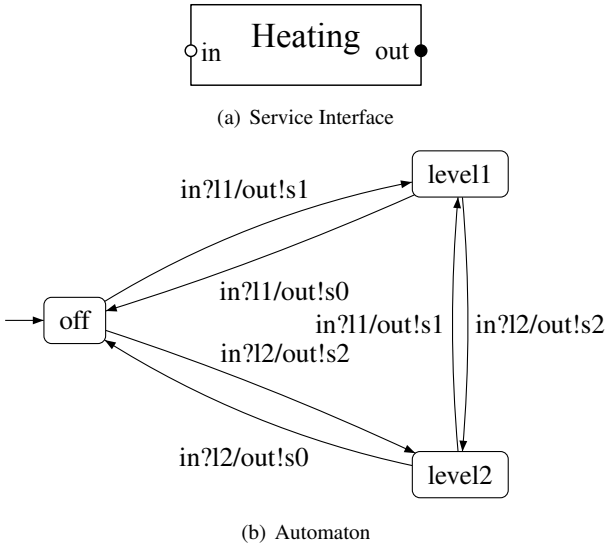
In the following, we introduce a notation for the specification of a single service, then both service models, the Service Diagram and the Service Network, are discussed in detail. Finally, we give an outline how to use these models for the construction of a Logical Architecture.

5.1 Service Specification

There are several techniques to specify the behavior of a service. In Section 4 services are formally defined by stream-processing functions. Since a service is a set of interaction patterns which give a precise relation between inputs and outputs, we propose to use modified I/O-automata [18] to specify a single service. An I/O-automaton $\mathcal{A}$ is completely defined by its set of states S, the initial state $s_0 \in S$, and the transition relation δ. A transition is denoted by

$$(s_1 \xrightarrow{in/out} s_2) \in \delta_{\mathcal{A}}, \quad \text{for } s_1, s_2 \in S.$$

A transition can be triggered in the state s_1 if all the input ports specified in the input pattern in have received the necessary input messages (denoted by $port?message$). The transition outputs data to different output ports specified in out (denoted by $port!message$), and puts the automaton into the state s_2. In other words, expression $port?message$ (resp. $port!message$) means that the message stream on the input (output) port $port$ is extended by the message $message$. Thus, the I/O-automaton constructively defines (infinite) input and output message streams as well as the relation between them. The automaton is partial in the sense that not for every input in every state there is a defined transition $t \in \delta_{\mathcal{A}}$. Since the behavior of the service is specified from the environment viewpoint (black-box view), the automaton is not allowed to have internal transitions (labeled with an empty input sequence).



(a) Service Interface

(b) Automaton

Figure 2. Specification of Service *Heating*

For example, Figure 2 shows a possible specification of the service *Heating* from our running example. This service describes a function that controls the heating of a seat. The user can switch between three states of the heating (*off*, *level 1* and *level 2*), sending one of the two messages (*l1* or *l2*) to the service. The service sends a corresponding message (*s0*, *s1* or *s2*) to the physical device responsible for the heating. The syntactic interface ($I \blacktriangleright O$) of this service is defined by I=$\{in\}$ and O=$\{out\}$. Figure 2(a) introduces a possible graphical notation. Figure 2(b) shows a behavior specification of the service. The I/O-automaton specifies a causality property between an infinite stream on the port in of input messages $\{l1, l2\}$ and an infinite stream on the port out of output messages $\{s0, s1, s2\}$.

5.2 Service Diagram

The last section showed how to 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 systems as observable from the environment when viewing the system as a black-box, i.e. the behavior is specified as a causal relation between input and output messages. 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 predication about 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 single services and three kinds of relationships between them, namely *refinement*, *aggregation*, and *dependency* (cp. Figure 3).

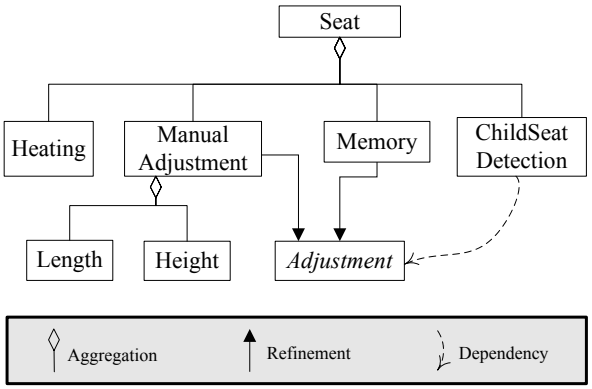


Figure 3. Service Diagram

Refinement Since a typical multi-functional system offers the 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*, since the partiality of a service allows to model a certain functionality at different levels of abstraction.

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 specified by that abstract service. Although, an abstract service is specified by an I/O-automaton it cannot be implemented directly, but rather must be refined by other services – it only helps to structure services and, particularly, dependencies between them.

The formal definition of an abstract service is based on the refinement relation between two services. Intuitively, an abstract service has fewer legal inputs and/or more defined outputs than the refining service. This means that for a certain input stream the set of corresponding output streams of the refining service R is a subset of the output streams of the abstract service A. Additionally, $Dom(A) \subseteq Dom(R)$ is required. In particular, R gives a more concrete/restricted specification of the function modeled by A since it acts more deterministically concerning its I/O relation. Our definition of refinement permits to change the number of ports and their types in a specific way. For the formal definition see [4].

For example, the service *Manual Adjustment* from Figure 3 refines the abstract service *Adjustment*. The Service *Adjustment* has only one input and one output port, and specifies an abstract seat adjustment functionality: It receives a message defining the movement direction ($d1$ or $d2$) and sends a corresponding motor control message m that encodes the direction. *Adjustment* only specifies an immediate causality between its input and output messages: the user gives a direction to the DCU – the seat moves in the given direction. *Manual Adjustment* gives a more restricted specification of the adjustment function – it separates between length and height movement. The refining service has two input and two output ports, for the length and height movement directions respectively. In the Service Diagram from Figure 3, service *Memory* is another refinement of *Adjustment*. It receives an input message and moves the seat to a saved position, according to the stored height and length. The abstract service *Adjustment* will not be implemented in the Service Network because its role is only to identify some common behavior between services *Manual Adjustment* and *Memory*. Both services have an important property in common – they are restricted due to the dependency relation between *Child Seat Detection* and *Adjustment* (see below).

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.

The *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. Intuitively, a sub-service specifies a sub-functionality of its super-service, in contrast to a refining service which refines the functionality of the whole refined service. Formally, a service S is called a *sub-service* of a (super-) service C, if C refines S and $Dom(S) \subseteq Dom(C)$. With this definition we require that the super-service has to be defined for all inputs for which its sub-service yielded a defined output as well. Additionally, the output streams of the super-service may be more restricted compared to the corresponding ones of the sub-service because of influences by other sub-services.

A super-service composed of several sub-services is called a *compound service*. Thus, according to the aggregation relation, we define the semantics of a compound service as a container of all *concurrently* operating sub-services. We do not specify the compound services by I/O-automata,

because their behavior can be reproduced from their sub-services using the well-defined semantics of the aggregation relation. Thus, the automaton of *Manual Adjustment* does not exist in the Service Diagram because it can be reproduced from the automata of sub-services *Length* and *Height*. In this example, the sub-services of the compound service are completely independent of each other – each of them can run without any impact on the other service. In other cases, 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 they are combined in one compound service. For example, since the service *Child Seat Detection* influences the services *Manual Adjustment* and *Memory*, we have to specify this dependency in addition to the aggregation relation between compound service *Seat* and sub-services *Heating*, *Manual Adjustment*, *Memory* and *Child Seat Detection*. See the following paragraph for further details.

Dependency Aggregation and abstract services represent hierarchical relations between services. Although the structure of services is essential for understanding the user functionality of a system, dependency relations among services also have significant implications in the development of a system. By *dependency relations*, we mean relations between services in a way that the operation of one service depends on those of other services. Although there are a lot of methodological significant dependency relations like *enables*, *modifies* or *needs*, the scope of this paper is to approach the specification technique of these relations rather than to completely enumerate them.

In our example (cp. Figure 3), the service *Child Seat Detection* modifies the behavior of the abstract service *Adjustment* and, hence, of *Manual Adjustment* and *Memory*. If *Child Seat Detection* detects a child seat mounted on the front seat, both adjustment services, *Manual Adjustment* and *Memory*, are prevented to move the front seat according to their modular specifications.

As already mentioned, relations 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). Also, the modular behavior specification of single services (in our case I/O-automata) are not modified in order to realize a dependency relation between them.

To specify these relations 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 predicate logic expressions over names of services, ports as well as access operations and specify dependencies between port values of different services in time intervals.

For example, the service *Child Seat Detection* (CSD) permanently receives a message (yes or no) from the environment through its port in whether a child seat is mounted or not. If the message is yes, service *Adjustment* (A) is not allowed to move the front seat, i.e. to send a message through its port out. The dependency between both services is specified by the following constraint:

$$\forall t \in \mathbb{N} : (CSD[in](t) = yes) \Rightarrow (A[out](t) = [\,])$$

This means, for every point in time, if message yes is received on the port in of service *Child Seat Detection*, no message is sent on the port out of service *Adjustment*.

As our example illustrates, abstract services implicitly impose their constraints on the refining services. All constraints on an abstract service have to hold for its refining services. For our example this means that the constraint on *Adjustment* has to hold for *Manual Adjustment* and *Memory*, too.

It should be noticed that the constraints do not modify the modular specification of services. They only specify the interplay between them which must be satisfied in the Service Network (see Section 5.3). In other words, constraints provide criteria to verify the models of the consecutive layer. Only those models which do not violate these constraints are the candidates for valid models.

With the Service Diagram we introduced an adequate model for the specification of the user-visible functional requirements of the system under consideration. Hereby, the basic ideas are to reduce the complexity of the overall functionality by describing each of its functionalities independently by simple services (by means of I/O-automata), arrange these services into a service hierarchy (by means of aggregation and abstraction relations) and specify relationships between these individual services that show how they influence each other (by means of dependency relations).

5.3 Service Network

So far, the focus lay on a consistent description of the behavior as it is observable at the outer boundaries of the system. However, now we address the question how the black-box behavior (specified in the Service Diagram) can be realized by a network (*Service Network*) of communicating entities.

As we are still not necessarily interested in complete descriptions of the system behavior, again services are used as basic building blocks. But in contrast to the Service Diagram, in a Service Network the services communicate with each other in order to realize the demanded behavior. Architectural concerns on the other hand like the mapping from

services to components and designing a hierarchical component architecture are not considered at this level but delayed to the following layer.

The structural relations which were dominating in the Service Diagram mainly served the structured gathering and understanding of the system functionality. Thus, they are of no relevance in the Service Network. More precisely, in a Service Network the services are not hierarchically arranged and there exist neither abstract nor compound services. As mentioned in Section 5.2, the behavior specified by an abstract service is completely realized by its refining services. Compound services were basically used in order to structure the functionality. Since their behavior can be reproduced from their sub-services (in consideration of the relevant dependencies), it is enough to implement the refining services or sub-services respectively.

In our example only the services *Child Seat Detection*, *Memory*, *Length*, *Height* and *Heating* remain in the Service Network (see Figure 4). The compound services *Manual Adjustment* or *Seat* as well as the abstract service *Adjustment* must not be considered in the Service Network.

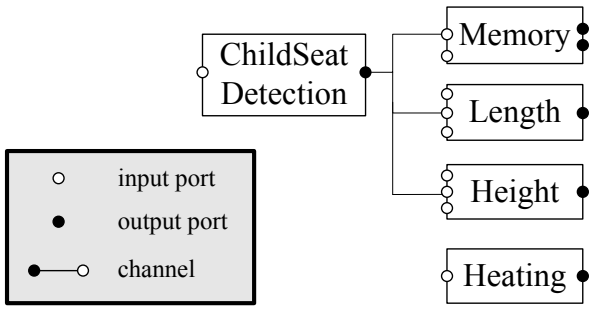


Figure 4. Service Network

However, it must be ensured that the resulting Service Network satisfies all dependencies in the Service Diagram. Therefore, dependency which are defined for abstract or compound services must be propagated to the refining services or sub-services and be reflected by their communication behavior.

The services of the Service Network refine the atomic services of the previous Service Diagram by adding communication behavior in order to realize the interplay between them. Thereby, each dependency relation of the Service Diagram results in a more or less complicated communication relation in the Service Network. This inter-service communication can be realized by adding internal channels in between the affected services. The syntactic interface and the behavioral semantics of the corresponding services must be extended accordingly. Sometimes, it can even be necessary or useful to introduce new services in order to realize the dependencies of the Service Diagram. In either case it must be assured that the behavior specified in the Service Diagram is completely realized in the Service Network.

In the following, both possibilities (adding new channels or new services) are explained in more detail. Note, that in both cases the syntactical interface of the overall system is not changed.

Internal Channels To establish communication between services, the interfaces of the services identified in the Service Diagram can be extended by additional internal ports. These can be used to link mutual depending services by internal channels. With *internal channels* we mean channels connecting two services, but not a service with the environment. Internal channels can only be attached to *internal ports*. In our example the functional dependency between *Child Seat Detection* and *Adjustment* (cp. Figure 3) is implemented by adding directed, internal channels connecting the service *Child Seat Detection* with all the services refining *Adjustment* (cp. Figure 4): the services *Length*, *Height*, and *Memory*. Via these channels the *Child Seat Detection* signalizes by a binary signal (*yes*/*no*) if a child seat is detected or not. Therefore, the interface of *Child Seat Detection* must be extended by an additional output port and the interfaces of *Length*, *Height* and *Memory* by respective input ports. Moreover, the behavior of these services must be adapted accordingly. The services *Length*, *Height* and *Memory* can execute their original behavior only if the messages received at the input ports connected to the service *Child Seat Detection* signalize that there is no child seat mounted.

Internal Services As mentioned before, it is sometimes useful to introduce new internal services. *Internal services* have only internal ports, i.e. ports which are not observable at the outer system boundaries. For example, priorities can be easily modeled in a Service Network that way. Let's assume that there would be the following additional dependency specified in the corresponding Service Diagram: The services *Length* and *Height* are mutual exclusive and *Length* has a higher priority than *Height*, i.e. if both length and height movement are demanded at the same time, the length movement will be executed. This relation can be implemented easily by adding an internal service *Multiplexer* which receives both input signals and decides if they are forwarded to the services *Length* and *Height* respectively.

Summarizing, the Service Network provides a specification of the overall system functionality (as specified in the services *and* constraints of the Service Diagram) only by means of communicating services. Thus, the Service Network constitutes a model that can be simulated easily. For example, all introduced concepts are directly supported by

the CASE tool AUTOFOCUS [21]. Besides, the Service Network can be used as starting base for the design of a Logical Architecture as introduced in the next section.

5.4 Logical Architecture

Changing the view from a pure black-box to a white-box view on the system the Service Network is the first step towards a logical system architecture. However, it is not dealt with architectural questions, and the specification of the system behavior is consistent but still incomplete. Thus, the most important task on this layer (*Logical Architecture*) is the totalization of the behavior. The behavior has to be defined completely and deterministically (i.e. the system provides a well-defined predictable output for each possible input sequence). As a consequence, the building blocks of the Logical Architecture are *components* with totally defined behavior in contrast to services. Therefore, the services are grouped together and mapped onto (hierarchical) components. Thereby, one or several services of the Service Network are related to one component of the Logical Architecture, i.e. a component can provide several services.

The second major design decisions we make on this layer is the design of an appropriate logical system architecture. The formation of the Logical Architecture can be influenced by different criteria, for example the communication relations of the Service Network, the hierarchical structure which has been developed in the Service Diagram, or certain non-functional requirements. Depending on which criteria are considered to be more important, the resulting Logical Architecture will turn out differently.

Though, the concrete definition of the Logical Architecture is essential for the development of a system, it is beyond the scope of this paper and subject of future work. Here, we just briefly sketched how the presented service models integrate in the overall software development process.

In this section, we presented three consecutive layers, namely the *Service Diagram*, the *Service Network* and the *Logical Architecture*, for the modeling of the functionality of a reactive multi-functional system. Note, that with the Logical Architecture the functionality of the system is completely described. Subsequent design models will not add any new functionality but only deal with the question how this functionality can be implemented adequately.

6 Related Work

The approach presented in this paper introduces an integrated model for both functionality specification and functionality hierarchical structuring as part of the layer framework. Thus, related work can be mainly found in three different areas: techniques for feature specification, formalization of feature models, and model based development.

Feature Specification A large number of contributions have been made over the past decade in order to specify multi-functional systems. Feature-oriented development methods, as for example Feature-Oriented Domain Analysis (FODA) [15] or Feature-Oriented Reuse Method (FORM) [16] identify, classify and structure features as well as interactions between them. FODA and FORM introduced a graphical tree-like notation that showed the hierarchical structure of features. Since the introduction of FODA by Kang et al. in 1990, many different kinds of graphical notations [11, 9, 12, 17] have been proposed to extend this original notation. However, these approaches only focus on the modeling of relationships between features, using uninterpreted features as the corresponding base concept. The second deficit of these methods is that the absence of a formal semantics of the graphical notations prevents an automatic analyze of them. In contrast, here the behavior of single features as well as the semantics of their relationships are specified.

Formalization of feature models The definition of a formal semantics for feature models is not new. In [2], Batory and O'Malley use grammars to specify feature models. Sun, Li et al. define in [23] a formal semantics for the feature modeling language using first-order logic. The formalization of feature models with propositional formulas goes back to the work by Mannion [19], in which logical expression can be developed for the model, using propositional connectives by modeling dependency between requirements. In [10], Czarnecki et al. argue that cardinality-based feature models can be interpreted as a special class of context-free grammars. Another approach to specifying multi-functional systems is introduced by van Lamsweerde et al. In [24, 25] they propose formal techniques and heuristics for detecting conflicts from specifications of goals (requirements) and their interactions specified in LTL. As mentioned in the latter paragraph, 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" [22]. In [8], Czarnecki and Antkiewicz recognize that features in a feature model are only merely symbols. They 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. Our work focuses on formalizing user requirements and their analyze in the early phases of the development process. The closest approach to our work is a theoretical framework introduced by Broy [3, 4]

where the notion of a service behavior is formally defined. This framework provides several techniques to specify and to combine features based on their behaviors. However, this quite theoretical approach does not cover several relevant methodological issues what our work focuses on (techniques for building of service models and for the specification of inter-service relations).

Model based development Another related area to our work is model-based development (e.g. [1]), which aims at modeling every important aspect of a software system. A compilable and deployable model is an abstract representation of a system which interacts with its environment. An important work in this area is the generative software development [7] introduced by Czarnecki. This system-family approach focuses on automating the creation of system-family members: a system can be automatically generated from a textual or graphical specification. However, this approach as well as approaches like [20, 22, 21] focus on the construction of a specific solution (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.

7 Conclusion and Future Work

The presented concepts can be roughly summarized as follows: We introduced two consecutive service models which allow to specify the functionality offered by a system in detail and shortly sketched how they can be integrated in a layered development process. The Service Diagram focuses on the modeling and structuring of the user-observable functionality while the consecutive model, the Service Network, concentrates on the communication relation between the set of all services. Going from a structured view of the functionality (represented by services) to a less abstract level where the focus is on the interaction between all services represents a natural way of engineering a system in a top-down fashion: At first we identify (observable) services and try to structure them into a hierarchical relation. After that we focus on how this hierarchy can internally be realized by inspecting the intercommunication between all relevant (possibly not directly observable) services. Lastly, the resulting Service Network serves as an architectural guideline for the subsequent construction of a Logical Architecture. In general, the introduced techniques are applicable in all domains where reactive systems are designed according to the desired functionality. In particular, they are not limited to the automotive domain.

Why did we concentrate to model the functionality of a system? In the domain of reactive, multi-functional systems, representing the functionality of a system means to describe precisely what a system should do, i.e. the functionality represents the essence of a system. Together with non-functional requirements we obtain a precise specification of the desired system already at a very abstract level. Note that this is essential since this level of abstraction of a system is the basis for changes due to the evolution or extension of the system with a maximum of re-use.

Both models, the Service Diagram and the Service Network, integrate into the requirements engineering process and bridge the gap between usually informally specified (functional) requirements and formal design models with a well-defined semantics. Our intension is not to replace the informal modeling techniques for the early requirements engineering process, but in our opinion informal models alone are not sufficient.

For a model-based development process, a seamless transition between the models of different abstraction layers is essential. The same notion of a service as the basic entity for describing functionality provides the basis for a smooth transition between both service models, even though the focus of the models is different.

The well-defined semantics of the introduced models allows to perform an automatic analysis of system properties. This allows to comfortably deal with problems such as feature interaction [6], resolving discrepancies between conflicting functions and verification of the system's behavior. The identification of all relevant domain-specific dependencies is part of our future work.

The concept of a service represents a suitable instrument to model single functionalities in a modular fashion. Together with a well-defined meaning for the composition of several services we are able to ascribe the specification of the overall system behavior to the specification of individual (sub-) functionalities. This helps to master the complexity of multi-functional systems.

So far, we have pretended to use the introduced techniques only to model single systems. But the principle of representing a (single) system as a set of related functions can easily be taken to model a family of related systems which share a set of functions. In turn, this requires to integrate concepts like variability and alternatives into our models and to define their semantics. For our service models, the notion of refinement and aggregation provides an adequate semantical basis to smoothly incorporate these concepts. By this, both models are suitable for the development of software product lines. We consider the extension of our concepts to be applicable in these areas as future work.

There are several issues which we did not point out in detail in this paper: E.g. we did not precisely specify the transition from the Service Diagram to the Service Network nor address the concrete form of dependencies together with a precise meaning. The definition of service-patterns for the

modeling of certain scenarios would enrich the degree of application in industry. Thus, such definitions and descriptions are a matter of future work as well.

References

[1] Model-driven architecture, 2004. http://www.omg.org/mda.

[2] 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.

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

[4] M. Broy. A theory for requirements specification and architecture design of multi-functional software systems. To appear.

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

[6] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Comput. Networks*, 41(1):115–141, 2003.

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

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

[9] K. Czarnecki and U. W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[10] 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.

[11] M. Griss, J. Favaro, and M. d'Alessandro. Integrating feature modeling with the RSEB. In *Proceedings of theFifthInternational Conference on Software Reuse*, pages 76–85, Vancouver, BC, Canada, 1998.

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

[13] 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 31.10.2006.

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

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

[16] 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.

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

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

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

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

[21] B. Schätz. Mastering the complexity of reactive systems: the autofocus approach. In *Formal methods for embedded distributed systems: how to master the complexity*, pages 215–258, Norwell, MA, USA, 2004. Kluwer Academic Publishers.

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

[23] J. Sun, H. Zhang, and H. Wang. Formal semantics and verification for feature modeling. In *ICECCS '05: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)*, pages 303–312, Washington, DC, USA, 2005. IEEE Computer Society.

[24] A. van Lamsweerde. Goal-oriented requirements engineering: A guided tour. In *RE '01: Proceedings of the 5th IEEE International Symposium on Requirements Engineering*, page 249. IEEE Computer Society, 2001.

[25] A. van Lamsweerde, E. Letier, and R. Darimont. Managing conflicts in goal-driven requirements engineering. *IEEE Trans. Softw. Eng.*, 24(11):908–926, 1998.