Institut für Informatik
der Technischen Universität München

A methodology for modeling usage behavior of multi-functional systems

Sabine Rittmann

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		Univ.-Prof. Dr. Uwe Baumgarten
Prüfer der Dissertation:		
	1.	Univ.-Prof. Dr. Dr. h.c. Manfred Broy
	2.	Prof. Dr. Martin Glinz, Universität Zürich / Schweiz

Die Dissertation wurde am 26.02.2008 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 02.09.2008 angenommen.

Abstract

Today we face a trend toward more and more complex systems. The complexity does not only arise because of distributed or heterogeneous hardware solutions. So called *multi-functional systems* which are characterized by a high degree of dependencies between functional units are also a major challenge. Often, the complex interactions between pieces of functionality (often referred to as *feature interaction*) are not understood and cause inconsistencies and unwanted behavior.

In order to handle this intricacy, new development techniques have to be created targeting at handling different aspects in isolation. This can be achieved by specifying the system functionality separately, i. e. with no structural and technical details involved. This functional model can then be mapped to the technical architecture. As the mere functionality of a system can be very complex again, both a local and a global view have to be provided. A *local view* describes pieces of functionality separately which reduces complexity and assures reuse. A *global view* integrates local views to form a larger (up to the whole) system functionality and sketches the "big picture".

Currently there is no methodology for specifying the functionality of a multi-functional system adequately. Most techniques mix up different aspects. They do not clearly separate the design of functionality and structure, or do not distinguish between interface behavior and realization, and between local system views and global system views. Especially the integration of functional specifications is not clearly understood. So-called (unwanted) feature interaction and dependencies between functionalities often pose major problems. Most approaches also lack a formal foundation. Another problem is the gap between the informal requirements phase (dealing with texts in natural language) and the formal design phase (dealing with models).

In this thesis, we introduce *concepts*, *suggestions for notational techniques*, and *methodological support* for designing usage behavior of multi-functional systems. Hereby, we only model the behavior as it can be observed at the system boundaries (*black box behavior*). The basic building blocks of the methodology are *services*. Services are pieces of (partial) behavior that relate system inputs to system outputs. The basic idea behind the approach is the following: modular services (*local view*) and the relationships between these services are captured. Based on the relationships between the services, the modular service specifications are combined step by step until the overall system functionality (*global view*) is obtained. Depending on the relationships pointing at a service, a service can be influenced differently (e. g. disabled or interrupted). The (modular) service specification has to be modified in order to handle these influences (e. g. it has to provide a special behavior while being disabled). This modification is done schematically and specific to the service relationships pointing at a service. We introduce so-called *standard control interfaces* which the modular service specification has to implement in order to handle the influences of the service relationships. Furthermore, we investigate how conflicting influences can be handled.

The contribution of this thesis is a methodology to formally specify the usage behavior of multi-functional systems. Our approach is a *model-based requirements engineering approach* enabling the stepwise transition from informal texts written in natural language to formal models of the system functionality. The result is a formal model of

the overall black box system functionality. During the formalization process, missing requirements can be detected. Furthermore, contradictories can be identified. Various applications of the system model can be thought of. For example, model-based testing can be applied to the formal models to test the correctness of the integrated behavior.

As far as the development process is concerned, the issues covered by this thesis are situated at the end of the requirements phase at the transition to the design phase. Furthermore, we focus on the specification of multi-functional systems.

Although our approach is not specific to a particular domain, we make use of a running example from the automotive domain in order to illustrate our concepts and to present the methodology.

Acknowledgements

My thanks go to **Prof. Dr. Dr. h.c. Manfred Broy** for having invited me to work in his research group and having provided me with the opportunity of writing this thesis. It was a pleasure to work within the great atmosphere of this chair, to be able to work together with industry, and to have insights in so many interesting topics. Furthermore, I would like to thank for comments on work presented in this thesis.

Additional thanks go to **Prof Dr. Martin Glinz** for also supervising my PhD thesis. I am very thankful for the detailed and constructive comments on how to improve this thesis.

Furthermore, I am grateful to **Markus Bechter, Florian Deißenböck, Alexander Harhurin, Judith Hartmann, Markus Herrmannsdörfer, Florian Hölzl, Johannes Grünbauer,** and **Birgit Penzenstadler** for fruitful and patient discussions and comments on drafts of this thesis.

A very special "thank you" also goes to **Silke Müller** and **Bernhard Schätz** for taking care of the chair in many regards!

I am also very grateful to my colleagues **Andreas Fleischmann, Eva Geisberger, Judith Hartmann, Markus Herrmannsdörfer, Michael Meisinger, Stefano Merenda**, and **Doris Wild**. I appreciated to work with you very much! I would also like to thank **Markus Pister, Wassiou Sitou,** and especially **Bernd Spanfelner** for always joking around, standing my sense of humor, and brightening my workdays.

Sincere thanks go to all of my **friends**. You made an excellent work life balance possible! Finally, I want to thank my family - especially my **Mum**, **Dad**, and **sister Marion**. You always lovingly supported and encouraged me in life. Thank you for all the things you enabled me to do! I love you.

"... for systems with a large number of internal states, it is easier, and more natural, to modularize the specification by means of features perceived by the customer." ([Davis, 1982])

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The topic of this thesis is a methodology for modeling the usage behavior of multi-functional systems. In this chapter we give a motivation for our approach (see Section 1.1, *Motivation*). The content of the thesis is sketched out in Section 1.2 (*Content of this thesis*). This includes a scoping regarding the system classes for which the approach is suited and a scoping regarding the position in the software development process. In Section 1.3 (*Contributions of this thesis*), we list the major contributions of this thesis. An identification of gaps in the current state of the art concerning the formal modeling of multi-functional systems is given in Section 1.4 (*State of the Art*). Finally we give the outline of the thesis (see Section 1.5, *Outline*).

Contents

1.1. Motivation

Innovative functions often are one of the key potentials to competitive advantage. This can for example be observed in the automotive domain [Frischkorn, 2004, Nelson and Prasad, 2003]. However, their merit will be limited, if they can not be developed efficiently. When it comes to distributed systems, difficulties result from the integration of functions into a network of complex functional dependencies usually being deployed on a highly distributed network of heterogeneous control units and bus systems.

There is the need for new development techniques which are able to deal with this intricacy. As the complexity stems from various sources (e. g. functional complexity, distribution, and heterogeneity) a promising approach for the development of

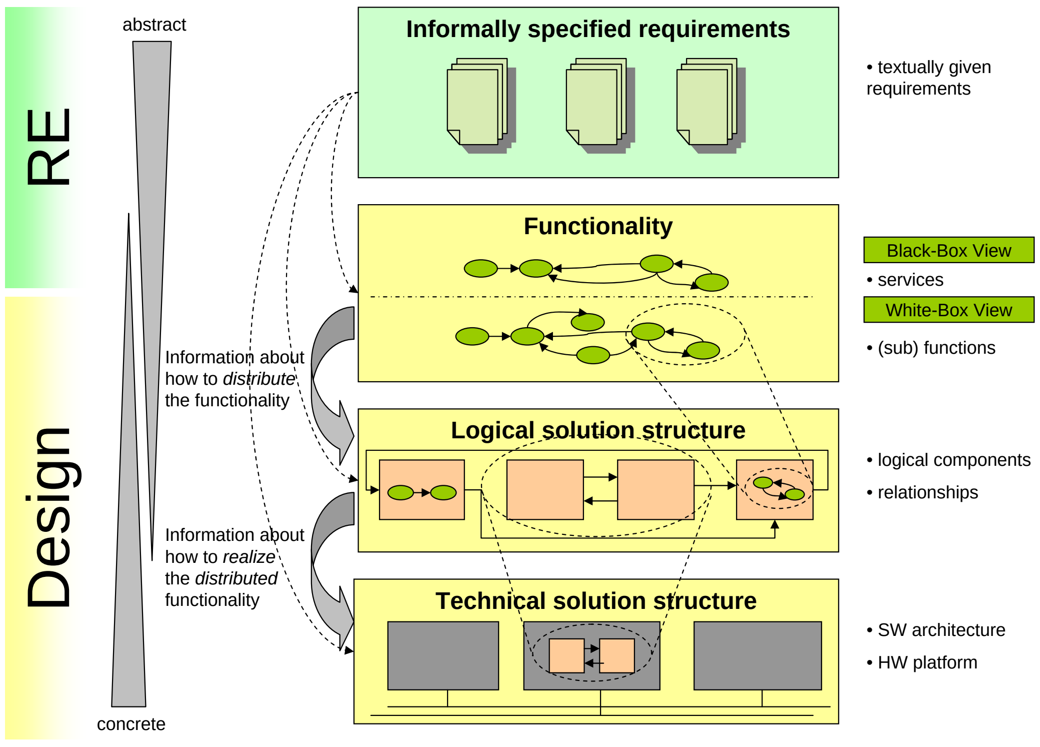
RE
Design
abstract
concrete
Informally specified requirements
• textually given requirements
Functionality
Black-Box View
• services
White-Box View
• (sub) functions
Information about how to *distribute* the functionality
Logical solution structure
• logical components
• relationships
Information about how to *realize* the *distributed* functionality
Technical solution structure
• SW architecture
• HW platform

Figure 1.1.: Development process along different levels of abstraction (schematic picture)

complex systems is by means of different abstraction levels.[1] Each of these abstraction levels deals with modeling certain aspects. The abstraction levels are based on each other starting with (abstract) high levels, adding more detail on each level, and leading to (concrete) low levels. During the transition from a higher level to a more concrete level, the model information is enriched; i. e. the completeness of the models - with regard to the implementation of the system - is increased. Thereby, the transition has to be correct: model information of abstract levels is inherited and must be obeyed and completely realized on the more concrete levels.

Figure 1.1 for example, shows three abstraction levels and how they relate to informally specified requirements. On the first level, the system functionality is modeled. The functional requirements (describing the system functionality) are input for this level. Details concerning the structuring of the system functionality into components and the technical structuring of a solution (by means of concrete hardware) are not taken into consideration. On the subsequent level, the model of the system functionality (as obtained in the previous level) is enriched with information about how to logically structure the model. The model is decomposed into logical components and grouped together in software architectures.[2] Finally, on the third level, the model is refined according to the information how to technically structure the solution. Al-

[1]The mobilsoft project [Mobilsoft, 2006] for example created a development process based on different abstraction levels for automotive systems. See [Hartmann et al., 2006c, Hartmann et al., 2006a, Hartmann et al., 2006b, Pfaller et al., 2006, Wild et al., 2006, Fleischmann et al., 2005, Rittmann et al., 2005] for more information.

[2]Ideas on structuring systems (into components/modules) go back to [Dijkstra, 1972] and [Parnas, 1972].

though the abstraction levels are ordered sequentially, the development process is usually performed iteratively as decisions made on more concrete levels have influence on more abstract levels.

A development process based on different levels of abstraction is especially advisable when developing a functionally complex system. So-called *multi-functional systems* [Deubler et al., 2004b] are such functionally complex systems. Functionally complex hereby does not refer to complex algorithms, but to a high degree of complex dependencies between (sub-) functionalities of the system behavior. For example, the remote unlocking of a modern car can be considered to be a multi-functional system. On pressing the remote car key, not only the doors are unlocked. Additionally, the emergency lights flash (in order to signal that the car has been opened), the interior lights go on, the driver's seat is adjusted according to a position saved before, etc. In order to provide this functionality, many functionalities have to work together (e. g. the functionalities responsible for controlling the emergency lights, interior lights, seat adjustment, etc.) and thus have to interact.

The complex interplay of functionalities often poses severe problems in practice. Dependencies and interferences between sub functionalities are not understood and thus not handled appropriately. Therefore, it is important to model the functional relationships within the system. As - for multi-functional systems - this alone is already very complex, issues concerning the logical and technical structuring of the system should not also be taken into consideration at once. Consequently, a development process along different levels of abstraction is needed to first model the pure system functionality.

When talking about system functionality, we can distinguish between the

- black box functionality/ behavior or usage behavior and the
- white box functionality/behavior

which is offered by a system (see again Figure 1.1). The black box behavior of a system is the functionality that can be observed at the system boundaries. Considering habitual language use, the black box behavior is often described by the terms "features" or "services" of a product. An example for a (piece of) black box behavior is the "comfort opening of a car window". The white box behavior of a system is a refined view onto the system behavior. It does not only describe the behavior from an external point of view, but also how the system functionality can be provided by the interplay of (only internally visible) sub functions. Therefore, a decomposition into sub functions which are called by each other and collaboratively establish the black box functionality is done. For example, the detection of the current of the window motor (in order to detect the end position) is part of the white box functionality which is not observable by the user (both human users and other (technical) systems communicating with the system under specification) at the system boundaries (if the sensor for detecting the resistance lies within the system boundaries).

The black box perspective on the system functionality is ideally used in the requirements engineering phase whereas the white box view is ideally part of the design phase. In the requirements engineering phase it has to be specified which functionalities the system under development has to provide and which relationships (e. g. mutual exclusion) are there between the system functionalities. Thus the following questions have to be answered:

- Which functionalities does my system have to offer to the user?
- Which relationships (or dependencies) between these functionalities should be realized?

Currently there is no methodology for specifying the system functionality (of a multi-functional system) adequately (see Sections 1.4, *State of the Art*, and 6, *Related Work*). Most techniques confuse different aspects. They do not clearly separate issues concerning the functionality, structure, and technical realization. Especially the distinction between the black box and white box specification of the system functionality is not done explicitly. The latter is important to clearly separate questions about what the system functionality should look like (*problem space* of the requirements engineering phase) from questions already concerning the realization of the system behavior (*solution space* of the design phase).

Also the integration of functional specifications is not clearly understood and often poses major problems. An overall (integrated) model of the system functionality (which is realized distributedly) in general is not created. Instead, textually given requirements are usually divided and assigned to components at the beginning of the design process and not reasoned about as a whole (component-based development approach). As a consequence, the interplay of components (resulting from the dependencies between the functionalities realized by the components) often poses problems. In this case, unwanted feature interaction [Zave and Jackson, 2000] can often be observed as a consequence. Moreover, most approaches lack a formal foundation.

Another problem is the transition between the requirements engineering phase which usually produces *informal* descriptions of the system under specification (e. g. textual descriptions) and the subsequent design phase which deals with *formal* models (e. g. automata, interaction models, system structure models). This "gap" often poses problems as the transition between the informal requirements phase and the formal (model-based) design phase is not understood and can not be performed seamlessly. Model-based requirements engineering is an upcoming concept which deals with introducing (semi-) formal models already to the requirements engineering phase and thus aims at bridging the aforementioned gap.

1.2. Content of this thesis

In the thesis at hand we introduce a methodology for modeling the *usage behavior* of *multi-functional systems*. We hereby are only interested in modeling the *functionality* of the system and do not take into consideration questions concerning distribution or technical realization. Considering Figure 1.1, we deal with the highest level of abstraction, the functional level. We present a *model-based requirements engineering approach*: Starting from informally (i. e. textually) given functional requirements, the requirements are specified more and more formally until a formal model of the overall black box functionality is obtained. We thus smoothen the transition between the informal requirements engineering phase and the formal design phase. In order to bridge the gap between these phases totally, the formal model of the black box functionality (which is the result of our approach) has to be refined to a formal model of the white box functionality. This is however not in the scope of this thesis.

The overall system functionality of medium to large scale systems usually can be very comprehensive. As far as habitual language usage is concerned, we often say that the system functionality is comprised of many (hundreds of) separate "services" or "features". But what exactly is a service or a feature? The terms service and feature do not have a uniform definition, respectively. Usually they vary from domain to domain [Kof, 2001, Meisinger and Rittmann, 2008, Salzmann and Schätz, 2003] and between different phases of the development process. In most cases the term "service"/"feature" refers to a characteristic functional property of a system. Preparatory work has been done in order to develop a formal service definition that can be used independently of the domain [Broy et al., 2007, Salzmann and Schätz, 2003, Schätz, 2002, Krüger, 2002]. These approaches define services by means of interaction patterns. Such an interaction sequence describes a piece of the system behavior by specifying the interplay between the system (or system entities) and its users. We base our approach on this intuition of the term service.

For the remainder of this text, we only make use of the term "service" (and do not use the term "feature" in order to avoid confusion). As the methodology uses *services* as basic building blocks, we speak of a *service-oriented approach* (related with *feature-oriented development*, *feature-driven development*, or *feature engineering* [Turner et al., 1999]).

In the thesis at hand, services are *pieces of partial black box behavior*. They are specified by partial input output mappings and capture the interaction between the system and its environment. Hereby, the initiative does not necessarily lie with the user. The system can start the interaction, as well. Services can be seen as *projections of the overall system behavior onto use cases*. Thus, services induce an aspect-oriented view onto the system architecture [Filman et al., 2004]. With the help of services, the overall system functionality can be structured into smaller pieces of behavior. This is very intuitive as the many stakeholders speak of services to have, change, or modify.

In order to reduce complexity, a modular approach is essential. The services are captured and specified modularly, i. e. as if they would exist in isolation. Then the services are combined. The collaborative interplay between the services establishes the overall system functionality.

However, there exist multiple *dependencies* between services within the integrated system. This is especially true for multi-functional systems which are - per definition - characterized by a high degree of interaction between (sub) functionalities. These dependencies change the modularly specified behavior of single services. This effect is known in literature under the keyword *feature interaction* (or *service interaction* in our nomenclature). A prominent definition of the term "feature interaction" is the following: "A feature interaction is some way in which a feature or features modify or influence another feature in defining the overall system behavior." [Zave, 2003] Examples for feature interaction (or: service interaction) are the abortion or the interruption of a service by another service. Although many feature interactions are wanted, unwanted feature interactions often cause severe problems in practice.

Feature interactions can occur because of different reasons. For example two sub functionalities might call an exclusively usable function at the same time. However, as we model the *black box behavior* of multi-functional systems, we are only interested in dependencies (relationships) between services which are visible at the system boundaries. Therefore, we only take into consideration feature interactions which occur because of user-visible relationships between services. Speaking more concretely, the abortion of a service by another service is in the focus of this thesis as it can be observed at the system boundaries. In contrast, the problem of two functionalities accessing an exclusively usable function at the same time is not looked at, as internal function calls can not be observed from a black box perspective.

Please note that we use the term "service" if we speak of a piece of *black box behavior* and the term "function" if we speak of behavior that might only be visible from a white box perspective.

As mentioned above, the projection of the overall system functionality results in "smaller" pieces of behavior - namely services. However, without the understanding of the dependencies between services, the projections of the system behavior would be highly nondeterministic. For example, the textual specification of the service "closing of a car window" could be as follows: "On pressing the toggle switch, the window has to be closed until the end position is reached." The textual specification of the child protection service could be: "If the child protection service is turned on, the windows can not be operated anymore." Only having the modular specifications without the relationship between them (namely that the child protection service prohibits the closing of the window), the system service "closing of a car window" would seem to be nondeterministic. In some cases the pressing of the toggle switch would result in closing the window (if the child protection service is switched off) and sometimes not (if the child protection service is switched on).

Above, we suggested to specify services modularly in order to reduce the system's complexity. However, these modular service specifications lack a treatment of service relationships (dependencies) as they model the service behavior in isolation and not in relation with other services. Consequently, feature interaction (being the result of dependencies between services) is not handled by the modular service specifications. The service specifications thus have to be modified according to the service relationships when they are combined to establish a more comprehensive behavior.

In the following subsection, we give a rough outline of our approach.

1.2.1. Rough outline of the approach

As starting point for our approach, we assume that the functional requirements are given almost completely. The focus of the approach is not the elicitation or derivation of requirements but the modeling of requirements. However, as a consequence of the modeling phase, missing - and therefore new, derived requirements - can be identified. (See also Section 1.2.3, *Scoping - Where in the software development process is the approach situated?*)

By functional requirement we denote the requirements describing the behavior of the system as defined in [Sommerville, 2004]: "These are statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations. In some cases, the functional requirements may also explicitly state what the system should not do."

Furthermore, we assume that they are given in an *informal*, textual form (i. e. in natural language).

The basic idea of the approach is the following iterative process:

As the functionality of the system can be very comprehensive, first single pieces of functionality (*modular atomic services*) which can be called by the user are identified in the functional requirements. Examples for such modular atomic services are: "turn on the radio", "adjustment of the height of the car's seat up", or "open the car remotely".

Furthermore, the syntactic interface of the system under development is specified: *Inputs* and *outputs* are identified and assigned to channels. Possible inputs and outputs are "press the radio button"[3] and "the height of the seat is increased", respectively.

The modular services are structured using hierarchical relations (*vertical service relationships*). On basis of these *service hierarchies*, dependencies between services (*horizontal service relationships*) are identified. Examples for such horizontal service relationship are: "mutual exclusion", "disable", and "interrupt".

In a next step the modular services which are at the bottom of the service hierarchy are specified by a formal model as if they were independent from each other. Afterward, the modular service specifications are combined step by step to form more comprehensive services. Depending on the horizontal service relationships pointing at a service, a service can be influenced, e. g. disabled. Thus, the modular service specification has to be adapted to handle these influences. For example, if a service can be disabled, its specification has to handle such a command and provide a respective behavior. This modification is done schematically (and specific to the horizontal service relationships) during the combination process: Each service specification has to implement *standard control interfaces* which are specific to the service relationships. By implementing the standard control interface, the services can handle the influences.

The combination process ends when the overall system functionality (being comprised of all atomic services) is constructed.

The main outcomes of the approach are the following:

- Both informal and formal modular service specifications which describe services (pieces of functionality) on different levels of granularity and which can be reused.
- A formal model of the overall integrated system functionality making all dependencies between functionalities/services explicit.
- A formal model of the overall system functionality.

[3]In fact, we will describe the inputs and outputs on a more abstract level in the methodology. For example, the input "driver wants to turn on the radio" abstracts from technical issues like a button or a toggle switch.

1.2.2. Scoping - What system classes is the approach for?

As mentioned above, we are interested in modeling multi-functional systems. Multi-functional systems are characterized by a high degree of dependencies between system (sub-) functionalities. We also speak of functionally complex systems in the remainder of this thesis whereas complex relates to the complex interplay of functionalities and not to complex algorithms. Of course the approach can also be applied to systems which are not functionally complex. However, this would add an inadequate level of complexity.

In order to describe the target domain of our approach, we briefly describe the system class we aim at. Computer systems can be classified according to different criteria. In the following we give four different possibilities to classify systems and classify our approach, respectively.

Classification according to transformation of data

Systems can be divided into three categories according to the way they transform input into output [Rechenberg and Pomberger, 1999]: *Transformational systems* transform input which is completely available to the system at the beginning of the execution into output. The output is not available until the end of the execution. The user (i. e. both human users and other systems) can not interact with the system during the execution and therefore can not influence the computation. *Interactive systems* interact and synchronize continuously with users during the execution. The interaction is determined by the system and not by the users. *Reactive systems* operate like interactive systems except that the interaction is determined by the user.

In our approach, the user interface is structured according to the (sub-) functionalities provided by the system interface and the dependencies between these (sub-) functionalities. As transformational systems provide a rather simple interface, they are not within the scope of this thesis. In this thesis we concentrate of *reactive* systems as the execution of the system can also be interrupted by the user and therefore the interface behavior is more complex to model.

Classification according to focus on control vs. data aspects

Systems can be classified according to their focus: The main focus of *technical systems* is to control technical processes. The main focus of *business systems* is on data storage and manipulation.

In this thesis, we develop an approach for *technical systems* as we do not intend to concentrate on aspects concerning data.

Classification according to the distribution aspect

Systems can be classified according to their kind of distribution: *Monolithic systems* are realized by one (logical or physical) component. *Distributed systems* are realized by components which are logically or physically distributed.

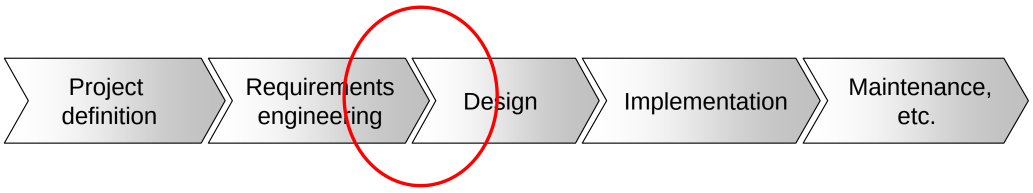

Figure 1.2.: Situation of the approach within the software development process (schematic picture)

In this thesis, we abstract from issues concerning distribution (and also from issues concerning the technical realization of the system). Consequently, the distinction between monolithic or distributed systems should not matter. However, as mentioned in Section 1.1 (*Motivation*), when distributing a system it is important to first create a model of the integrated system functionality. Against this background and due to the fact that a monolithic system can be seen as a special case of a distributed system, we focus on *distributed systems* in this thesis.

Classification according to variation in time

Considering possibilities at what point in time "relevant" activities are allowed to happen during the system run, the following classification can be done: In *time-continuous systems*, relevant activities can evolve continuously. In *time-discrete systems*, the time line is divided into not necessarily but possibly equidistant intervals. Relevant activities only occur at points of time of the time grid.

For our approach we focus on *time-discrete systems*.[4]

1.2.3. Scoping - Where in the software development process is the approach situated?

As far as a model-based software development process is concerned our approach can be situated at the transition of the requirements engineering phase and the design phase (see Figure 1.2). In order to explain this statement, we have a look at the typical tasks of the requirements engineering and design phase, respectively.

Tasks of the requirements engineering phase

The typical tasks of the requirements engineering phase are (adapted from [Sommerville, 2004]):

- requirements elicitation
- requirements specification/documentation
- requirements management
- structuring of requirements

[4]The modification of our approach to also suit the other system classes is considered to be future work. See also Section 7 (*Summary, evaluation, and outlook*).

- validation of requirements (e. g. analysis, simulation)
- verification of requirements (e. g. consistency checks)

In the elicitation activities, requirements are elaborated. This includes both the initial elaboration of requirements and the derivation of further requirements. The documentation aims at the (formal or informal) specification of the requirements. Management activities deal with how to manage the requirements. Structuring of the requirements is important for reasons of clarity. During the validation, the requirements are investigated if they really demand the intended system (behavior). The verification of requirements aims at identifying contradictories.

In our approach, mainly the following activities are covered: *requirements elicitation, requirements specification/documentation*, and *structuring of requirements*. We specify the functionality of the system by formal models. In case that not all of the information needed for creating the models is given, missing requirements can be detected and therefore implicitly elicited. The specification/documentation is explicitly done when modeling the services. As mentioned above, the system functionality is structured into service hierarchies which results in a structured view onto the user functionality.

As a side effect of the formal foundation of the introduced concepts, validation and verification can be done based on the models. For example models can be simulated in order to validate the specified functionality and consistency checks can be performed in order to verify the specification. However, this is not in the focus of the thesis at hand. Issues concerning the management of requirements are not taken into consideration.

Tasks of the design phase

The tasks of the design phase are the creation of (formal) models containing different views onto the system under development (and the integration of these models) [Broy and Steinbrüggen, 2004]:

- behavioral view
- state view
- structural view (architecture design)
- data view (data flow, data type definition)

As far as the approach to be presented is concerned, formal models for all of these views are created. The behavioral view represents the system functionality. States of the services are made explicit, therefore a state view is also provided. The structural view divides the system into user-visible services and the relationships/connections between these services. Data being the input and output of the system and needed to store information persistently are specified, too.

1.2.4. Scoping - What issues are not covered by the approach?

The thesis at hand presents a methodology for modeling the usage behavior of multi-functional systems. Of course, this is a very comprehensive topic. Not all interesting questions related to this topic can be handled in a single thesis. The following issues are not covered by our methodology and are subject to future work (see also Section 7.3, *Outlook*):

The aim of this thesis is to explore the concept of services as far as possible. However, we do not want to claim that services are the best solution for all abstraction levels of large-scale systems. For a pragmatic approach, further abstraction techniques should be combined with the presented approach and evaluated by means of comprehensive case studies. However, this is not in the focus of this thesis.

The presented approach is only appropriate for modeling systems with a trivial set of persistent data. In order to handle systems having a more comprehensive set of persistent data, the approach has to be modified appropriately.

In the thesis at hand we make use of a very simple time semantics which is not appropriate for modeling the timing behavior of real systems. The approach can be modified accordingly, however the modifications make it more complex. We will outline how the approach can be changed toward an adequate time semantics in Section 7.2.2 (*Disadvantages of the approach*).

Product line development aims at maximizing reuse. For example, see [Böckle et al., 2004] and [Clements and Northrop, 2002] for a general introduction on product line development. The basic idea is to first identify similarities and differences of products within the same domain. On basis of a common platform, new products are then developed by adding "building blocks" to this platform. In our approach we do not take into consideration issues concerning product line development. Furthermore, instantiation (e. g. instantiation of parts of models) is not covered either.

Besides functional requirements, non-functional requirements (e. g. covering reliability, security, performance issues) are very important when specifying the intended system behavior. However, this topic is very complex and would go beyond the scope of this thesis.

Other issues like the specification of continuous behavior which are not covered by our approach result from the target system type (see the preceding subsection).

1.3. Contributions of this thesis

In this section, we briefly list the main contributions of this thesis. A more detailed discussion on the contributions is given in Section 7.2 (*Evaluation*).

The scientific contribution of this work is the introduction of a methodology for the compositional, formal specification of multi-functional systems. The model-based approach is situated at the transition from the requirements and design phases and thus bridges the gap between detailed functional requirements and a logical system architecture.

The thesis bases on the following approaches: self-contained, hierarchical services which communicate with each other by means of typed, directed channels are used to describe the system structure. The behavior of these components is modeled by a Statechart variant. The decomposition of the system functionality is done by so-called service graphs. The hierarchical dimension of service graphs is a well known way of structuring functionality.

The original contribution of this thesis is a methodology which enables the systematic development of multi-functional systems by combining smaller pieces of functionality, namely services. The combination is done on basis of service relationships which describe the dependencies and influences between services. During the composition of services to more comprehensive services, the transition of informal requirements to formal specifications is done.

Methodology We present a methodology describing how to proceed when modeling the black box functionality of multi-functional systems. To that end we describe steps (*activities*) leading to well-defined products (*artifacts*) and an ordering on them. The artifacts serve as milestones in the development process.

The process allows a comprehensive *reuse of artifacts*. As all of the artifacts of our approach are modeled as modularly as possible, not only the outcomes of the methodology but also intermediate results can be reused. For example, informal and formal specifications of services of an arbitrary level of granularity can be used for the specification of further systems, again.

By relating services, dependencies between services are made explicit. Thus the complex interplay is understood. Feature interaction (service interaction) can be handled systematically. This enables an *understanding of the complex dependencies* within the system functionality.

Although the elicitation of missing requirements is not in the scope of our thesis, a positive side effect of our approach is that missing requirements can be detected (*detection of missing requirements*). This is due to the model-based character of our approach. As we will see later, we can check if all the information required by the models is covered by the requirements. If not, further requirements containing the missing information have to be elicited.

As another consequence of the modular approach, parts of the system functionality can be specified in isolation and then integrated. Thus, *distributed development* is possible. This is especially important concerning the producer-supplier-relationship. As only the interface specification of the sub systems has to be known, the internal realization of sub systems can be hidden. The intellectual property (IP) of a company therefore is protected.

Formally founded concepts In our methodology, modular service specifications are modified to handle influences by other services, e. g. to handle an interrupt command. The modular service specifications have to implement standard control interfaces in order to handle these service interactions. For each standard control interface we present a schema how to systematically realize it. These schemata are formally founded and given a precise semantics. Consequently, the meaning of the concepts is defined without ambiguities.

Notational techniques For the specification of the system functionality adequate notional techniques are suggested and introduced for each step of the methodology. Wherever possible, standard notations (like State Charts) are used. Additionally, the introduced notational techniques are very intuitive and easy to learn.

1.4. State of the Art

We compare our work to a selection of related approaches in Section 6 (*Related Work*). At this place we shortly identify gaps in the state of the art concerning the formalization of functional requirements and show how our approach closes them.

The idea of a continuous model-based development process is relatively new. Most work centers on a (distinguished) phase (or topic) in the development phase. The transition between development phases is often neglected. For example, usually, requirements engineering approaches are informal, whereas design approaches are formal. We investigate gaps in these two areas in Sections 1.4.1 (*Requirements engineering approaches*) and 1.4.2 (*Formal modeling of the system functionality*). In our methodology, modular service specifications are systematically adapted to handle feature interaction. Thus, we also investigate the state of the art in research on feature interaction (see Section 1.4.3, *Work on feature interaction*). Many work on feature modeling, especially decomposing the system functionality into trees, and identifying relationships between features, has been done in the area of product line development. In Section 1.4.4 (*Feature modeling within product line development*) we identify gaps in this context.

1.4.1. Requirements engineering approaches

Requirements engineering approaches can be roughly classified according to their basic concepts, for example: Use Case/Scenario-based approaches, goal-based approaches, linguistic methods, and model-based requirements engineering.

Use case-/Scenario-based approaches

Use cases are commonly used in practice to capture the functionality of a system. They describe the tasks which the system has to provide from an external point of view and thus are helpful to determine the system boundaries. So-called "actors" are entities which interact with the system. A use case is an abstract description of all actions and focuses on completeness whereas scenarios usually are concrete instances of use cases. They focus on understandability by describing concrete sequences of interactions between the system and actors. The most prominent representative of use cases probably are UML Use Case Diagrams [OMG, 2003]. They are usually accompanied by structured, textual descriptions. Although templates for use case diagrams are structured into entries like participating actors, precondition, flow of events, and post condition no schema on how to obtain (which kind of) formal models out of the textual descriptions is given.

Model-based use case approaches intend to overcome this problem. They enrich use cases by formal semantics. In [Dano et al., 1997] petri nets are used to give use cases a precise meaning. Others make use of sequence-based languages like MSCs [ITU-T, 1996] or Live Sequence Charts [Damm and Harel, 2001] (enhancing MSCs by modality, anti-scenarios, etc.). The tool AutoRAID [AutoRAID, 2007] for example makes use of sequence diagrams, called Extended Event Traces, to describe functional requirements. Although being more formal than textual use case descriptions, these approaches do not take into account dependencies between functionalities.

As far as dependencies between functionalities are concerned, most (informal and formal) use case approaches only provide "extends" and "uses" relationships. The first one indicates that an extended use case may include the behavior which is specified by the extending use case. The second relationship is used to capture common behavior of use cases. It describes that a use case always includes the behavior of another use case. However, no formal semantics of these relationships is given. Other dependencies, capturing feature interaction are not provided. Furthermore, the user can use stereotypes to informally define further relationships.

Although often considered to be intuitive, these notions suffer from the following disadvantages: Different interpretations of the relationships are possible as their meaning is not precisely defined. Consequently, we have ambiguous specifications. This problem already occurs when only looking at two pieces of functionality. The problem explodes when more services (up to the overall system functionality) is looked at. The interplay of several functionalities and transitive dependencies pose severe problems and are not understood.

Goal-oriented requirements engineering

Goal-oriented requirements engineering approaches use goals for all requirements engineering tasks, i. e. elicitation, verification, documentation, etc. Hereby, a goal describes the stakeholders' intentions and desired system properties. Goals can be situated on different levels of abstraction. For example both "The system shall serve more passengers." and "Keep ATM card after three wrong password entries." are considered to be goals. Usually goal-oriented approaches structure the system hierarchically. A goal is broken down into sub goals and refined by "AND" and "OR" relationships. The AND relationship specifies that all sub goals have to be fulfilled for the fulfillment of a decomposed goal. The OR relationships states that it is enough if one sub goal is fulfilled. Furthermore, conflicts between goals are modeled.

One prominent representative of goal-oriented approaches is KAOS [Darimont et al., 1997]. Goals are classified according to types: Functional vs. non-functional goals, soft goals (the satisfction of which can be achieved through verification) vs. hard goals, system state (positive, negative, alternative, etc.) and others. Goals are refined until they can be assigned to agents within the system or the environment. KAOS also provides the possibility to model goals formally by means of a real-time linear temporal logic. However this is suggested only for critical aspects of the system.

Although formal goal-oriented approaches overcome the problems of informality, they still suffer the problems of not capturing feature interaction. Usually the only functional dependencies which are specified are AND and OR refinements of goals and goal conflicts.

Linguistic approaches

Linguistic approaches try to overcome the problem of imprecise, ambiguous textual specifications and to create formal models out of text.

Chris Rupp [Rupp, 2007] introduces a set of rules for the specification of requirements in natural language. For example, incomplete process words have to be completed. The requirement "The system is supposed to record the loss of data." lacks the information to whom it shall report the loss, how this should be done, etc. Superlatives have to be described further. The requirement "A faster memory has to be used." for example does not contain the information how fast this actually is. The approach by Rupp aims at informally but precisely specifying requirements. However, they do not target at bridging the gap between the requirements engineering phase and the design phase. Thus the introduced rules are not motivated by design models.

The rules by Chris Rupp are not implementable. Other approaches automatically analyze natural language to obtain design models. *Lexical approaches* analyze words and sequences of words and produce lists of key words based on frequency (for example Abstfinder [Goldin and Berry, 1997]). *Syntactic analyses* result in entity relationship diagrams or object-oriented models of natural language. Work by Abbott [Abbott, 1983] for example transforms nouns into data types, verbs into operations, etc. In [Kof, 2005] domain-specific ontologies, i. e. terms and relations between these terms, are extracted from natural language documents. *Semantic analyses* are based on syntactic approaches and aim at investigating the semantic role of words or parts of sentences (see for example Fillmore's Case Grammar [Fillmore, 1971]).

The above mentioned approaches produce models out of natural language. However, the result is not an executable model of the system functionality. Especially, the problem of feature interaction is not handled. Mostly, terms and relations between terms are identified.

Model-based requirements engineering approaches

Model-based requirements engineering is a quite young research direction. It aims at bridging the gap between the informal requirements engineering phase and the formal design phase.[5] Above, we already mentioned model-based use case approaches. The tool AutoRAID allows for the motivation of design model elements due to given requirements. For example, requirements can motivate components or states. However, requirements are only informally linked to single design model elements. In [Damas et al., 2006] an approach is introduced which specifies functional requirements by Message Sequence Charts (MSCs). The Message Sequence Charts are transformed into a labeled transition system (LTS) which is basically an automata dialect. Goals are inferred inductively from MSC scenarios.

[5]In Section 6.1 (*Model-based requirements engineering*) we compare our approach with model-based requirements engineering approaches.

Although model-based requirements engineering approaches have the same basic goal as our work (to systematically obtain formal models out of text) they do not pay enough attention at capturing functional dependencies appropriately. Furthermore they do not investigate how functional dependencies effect modular functional specifications.

1.4.2. Formal modeling of the system functionality

There exist many approaches which are concerned with formally modeling the system functionality. Function trees for example capture functional entities which are related to each other by different relationships. These relationships are usually "call" or "contain" relationships. Jackson diagrams describe structural and sequence-based relationships. The sequential execution of internal functions can also be specified by Jackson diagrams.

In [Deubler, 200x] service relationships are identified and formally specified. Although the aim of [Deubler, 200x] is to formalize service relationships from a pure usage view this goal is not achieved due to our opinion. For example relationships like "trigger" and "call" are introduced which represent the call of a service with or without feedback by the called service. As mentioned in Section 1.1 (*Motivation*) these relationships are not adequate for the requirements engineering phase as they already contain information about the realization of the black box system behavior. Furthermore, no methodological support for the formal specification of the system functionality is given.

In [Broy, 2007] service relationships (like the sub service relationship) and service refinement are formally defined. We make use of this formal work as basis for our approach (see Section A, *Embedding into a theoretical framework*). [Broy, 2007] aims at developing a formal theory for service-oriented requirements engineering rather than providing pragmatic, methodological support.

In [Schätz, 2007] the author also structures the system functionality hierarchically into modular pieces which are formally specified. The formal specifications are then combined by alternative combination (OR relationship) or parallel composition (AND relationship). However, no further relationships are identified to handle feature interaction. This is not in the scope of that work.

The CASE tool AutoFOCUS 2 [Huber et al., 1997] from which we adopt the notational techniques for our methodology provides different diagram types to model different views onto the system: System Structure Diagrams are used to model the syntactic aspects of the software-to-be like its architecture. State Transition Diagrams are an automata dialect to capture the local behavior of components. Extended Event Traces specify the interaction between system entities and between the system and its environment. Furthermore, Data Type Definitions can be used to define complex data types. Consistency checks between these different views onto the system are also implemented. Although AutoFOCUS 2 is connected to the requirements engineering tool AutoRAID [AutoRAID, 2007], only informal links (like "motivate") can be established between informal requirements and formal model elements.

Although many formal approaches for the modeling of the system functionality exist, most of them are not appropriate for modeling the system functionality from a requirements engineering point of view. As mentioned above, relationships like "call", "trigger", or "use" stem from the implementation and are motivated from a technical point of view. Moreover, most of these approaches are too formal and not intuitive enough for the requirements engineering phase. Furthermore, these approaches do not take into account how feature interaction effects modular, functional specifications.

1.4.3. Work on feature interaction

The most prominent definition of the term feature definition as mentioned in Section 1.2 (*Content of this thesis*) and much research around this topic comes from the telecommunication domain. See for example the international series of Feature Interaction Workshops or [Calder et al., 2003] for an overview of work on feature interaction in the telecommunication domain. In this domain, features are considered to be components of additional functionality and thus can be seen as our services. Examples for features are "call forwarding capability" or "ring back when free".

But also other domains such as computer aided design [Perng and Chang, 1997], process planning [Hwang and Miller, 1995], and in the area of embedded systems [Metzger, 2004] are affected by feature interaction. Research generally includes how to constructively avoid feature interaction or how to identify scenarios in which unwanted feature interaction might potentially occur or does occur and how to resolve these interactions. The approaches try to tackle the problem differently. Some make use of well-known techniques like state machines (see for example work by Braithwaite ant Atlee [Braithwaite and Atlee, 1994]), SDL (see [Kelly et al., 1995]), or LTL (see [Felty and Namjoshi, 2000]) during design time. Others deal with feature interaction by detecting and resolving it during runtime by (see for example [Marples and Magill, 1998, Homayoon and Singh, 1988, Aggoun and Combes, 1997, Reiff, 2000]).

The problem with existing approaches is that most of them are tailored to the telecommunication domain and not applicable to other systems. For example the prominent approach by Zave and Jackson (see Section 6.2.1, *Services in the telecommunication domain - Distributed Feature Composition (DFC)*) introduces a virtual architecture specific to the telecommunication domain. A precedence relation, i. e. a partial ordering of the features, is made available to the router in order to avoid unwanted feature interactions.

Many other approaches deal with feature interaction that is caused by shared resources like for example [Lorentsen et al., 2001]. However, on our level of abstraction we only want to look at the functional dependencies that can be observed at the system boundaries. In [Shehata et al., 2004] the term feature interaction is defined even more general and also takes into consideration non-functional requirements (again including shared resources). As mentioned above, a great variety of approaches is concerned with identifying and resolving feature interaction instead of systematically combining modular, reusable services to handle feature interaction.

1.4.4. Feature modeling within product line development

Much work on structuring the system (behavior) into "features" (or "services" in our setting) and identifying relationships between them exists in the area of product line development. FODA [Kang et al., 1990] introduces the well-known FODA trees ("Feature Model") which hierarchically decompose a system. However, FODA trees do not only decompose the system functionality, but also structure technical aspects.

Although there is a variety of other approaches [Kang et al., 1990, Kang et al., 1998, Kang et al., 2002b, Fey et al., 2002, Jacobson et al., 1997, Lee et al., 2000, Czarnecki, 1998] each of which defines service relationships - some informal without precise semantics, some formal - they are all domain-specific. The main aim of product line development is to maximize reuse. To that end commonalities and differences between systems in terms of common and distinct features are identified. Usually, feature models are AND/OR graphs. Features are structured hierarchically by means of composition, specialization, and implementation relationships. Features themselves can be declared to be optional, obligatory, or alternative. Furthermore, these graphs are enriched with "excludes" or "requires" relationships.

In addition to the domain specificness, another fact makes these approaches inappropriate for our work: often implementation alternatives specifying alternative implementations for functional features are also encoded in the feature models. Thereby, these implementation relationships are not always "just" tracing links but intertwined with the pure functionality.

1.4.5. Summary

This thesis introduces a model-based requirements engineering approach which aims at modeling the functionality of multi-functional systems from a black box view. Thus it aims at bridging the gap between the informal requirements engineering phase and the formal design phase. Focus is given on how dependencies between functionalities effect modular functional specifications (feature interaction).

In the paragraphs above we described the current state of the art in requirements engineering approaches, formal modeling of the system functionality, work on feature interaction, and feature modeling within product line development with regard to our work.

Typical requirements engineering approaches are usually informal in order to be intuitive. The problem which arises is that the semantics of their concepts are not formally founded and thus imprecise. The relationships between functionalities are usually motivated by a specific aim, for example by the refinement of goals (AND/OR relationships in goal-oriented requirements engineering approaches) or the reduction of specification effort (extends/includes relationships in UML Use Case Diagrams). However, relationships capturing feature interaction are not introduced. Linguistic approaches produce design models like entity relationship diagrams but do not model the system functionality. Model-based requirements engineering approaches do not investigate the effects of feature interaction.

Approaches concerning the formal modeling of the system functionality overcome the problems of imprecise semantics. However they are not appropriate for the requirements engineering phase which involves people (e. g. customers) with no formal background. Additionally, the relationships introduced are mainly motivated from a technical perspective rather than from the usage view.

Work on feature interaction does take into account the effects of a function onto another function. However, these approaches usually are also very focused on a technical view onto the system. For example, they investigate feature interaction which arises because of a concurrent access to shared resources. A pure view onto the system from a black box perspective, as needed during requirements engineering, is not given.

Other approaches like feature modeling within product line development or work on feature interaction within the telecommunication area is tailored to a specific domain and not appropriate for a domain-independent approach. Moreover, many approaches miss methodological support.

To put it in a nutshell: The area of model-based requirements engineering is a relatively young field of research. Most approaches are either informal or formal depending on the target group. The idea of hierarchically structuring the system functionality in smaller pieces is not new. However, the identification of differentiated service relationships as appropriate from a usage view, the stepwise combination of services on basis of these dependencies, and the systematic adaption of the service behavior during the combination process is new to the best of our knowledge.

1.5. Outline

This thesis is organized as follows:

In order to illustrate our concepts and to present the methodology we make us of a running example: the power seat control system as can be found in modern cars. In Chapter 2 (*Running example*) the functional requirements of this system are given in an informal (textual) form.

Chapter 3 (*Notational techniques (Overview)*) introduces notational techniques which are used in the methodology to be presented.

In Chapter 4 (*Methodology*) we introduce our approach. The iterative process for modeling the black box behavior of multi-functional systems is described step by step. After a description what we assume to be the starting point of our methodology (informally given, textual requirements), each step is presented in a separate section, respectively. The chapter is concluded by the description of the result of our approach (formal model of the overall system functionality) and some further considerations.

The basic idea behind our approach is the following: Modular service specifications are combined on basis of their service relationships. In Chapter 5 (*Extension of basic service relationships*) we again take a detailed look onto service relationships.

In Chapter 6 (*Related Work*) we list related work and compare it to our approach.

Chapter 7 (*Summary, evaluation, and outlook*) summarizes the approach. The methodological steps are revisited again. Furthermore, an evaluation of the methodology is done. Its advantages and disadvantages are discussed. Finally, an outlook on future work is given.

Also find a glossary at the end of the thesis with the most important key words used in our methodology. In the appendix (Appendix A, *Embedding into a theoretical framework*), we embed our approach into a formal framework.

Chapter 2

Running example

In order to explain our concepts and the introduced methodology we make use of a case study from the automotive domain: the power seat control system. In this chapter we list the textual (informally given), functional requirements for this system. The requirements of the power seat control are a modified and simplified version of the requirements presented in [Houdek and Paech, 2002]. Among others, we make the following simplifications: Our seat adjustment system is only available for the driver's seat. Furthermore, only one position can be saved and called.

To be able to reference the requirements within the proceeding chapters, the requirements are numbered.

First, we list general requirements of the power seat control system (see Section 2.1, *General requirements*). Then we list the requirements for the manual adjustment of the seat and the adjustment by memory, respectively (see Sections 2.2, *Requirements for the manual adjustment* and 2.3, *Requirements for the adjustment by memory*).

Contents

2.1. General requirements

1. The driver can adjust his/her seat according to his/her requirements. The driver has the possibility to do the following adjustments:

    a) Adjustment of the angle of the back

        i. forwards

        ii. backwards

    b) Adjustment of the distance between the seat and the steering wheel

        i. increasing
        ii. decreasing
    c) Adjustment of the height of the rear area of the seat
        i. up
        ii. down
    d) Adjustment of the height of the front area of the seat
        i. up
        ii. down

2. In order to adjust the seat the respective motors are controlled.
3. There are two possibilities to adjust the position of the seat: The adjustment of the position can either be carried out manually (*manual adjustment*), e. g. by means of a toggle switch, or automatically by triggering the *memory functionality*.
4. The adjustment (manual adjustment or memory functionality) that was selected last determines the movement of the seat, respectively.

2.2. Requirements for the manual adjustment

5. The manual adjustment of the seat is only possible if the front door is open.
6. The manual adjustment can be triggered by pressing the toggle switches (one for each adjustment: back, distance, rear area, front area) which are attached to the seat.
7. The manual adjustment of the seat is carried out as long as the user request is active, respectively.
8. For each adjustment (back, distance, rear area, and front area) only one direction of movement (e. g. up or down) can be carried out at once in case of the manual adjustment.
9. At most two of the adjustments can be carried out simultaneously.
10. In case the battery power is too low (lower than 10V) to perform the manual functionality
    a) the seat is not adjusted.
    b) If need be the currently active adjustment is aborted.
    c) Additionally, the error message err_bat_low_man is sent on the bus.

2.3. Requirements for the adjustment by memory

11. The memory functionality can only be executed as long as the car's velocity is not too high (not higher than 5km/h). If need be, the adjustment of the seat is

aborted immediately.

12. The memory functionality can be triggered either by pressing the button attached inside the driver's door (so-called *seat button*) or by opening the car remotely with a *car key*.[1] (Please note that our requirements only cover the retrieval of the seat position and not its saving as in [Houdek and Paech, 2002] this functionality is part of the user management system.)

2.3.1. Requirements for the memory functionality triggered by the seat button

13. If the memory functionality is triggered by pressing the *seat button*, it is assumed that the driver is already sitting on the seat. In order to make the adjustment of the seat as comfortable as possible
    a) first those directions are carried out that cause a relaxation: moving the back backwards, increasing the distance, moving the front area of the seat down, moving the rear area of the seat down.
    b) Afterward, the opposite directions are carried out.
    c) At most two directions can be carried out simultaneously.
14. In case the battery power is too low (lower than 10V) to perform the memory functionality (which was triggered by the seat button)
    - the seat is not adjusted.
    - If need be, the currently active adjustment is aborted.
    - Instead, the error message err_bat_low_seat_button is sent on the bus.

2.3.2. Requirements for the memory functionality triggered by the car key

15. If the memory functionality is triggered by pressing the *car key*, it is assumed that the driver is outside the car (and not sitting on the seat). In order to move the seat to the desired position as soon as possible, all directions are carried out simultaneously.
16. In case the battery power is too low (lower than 10V) to perform the memory functionality (which was triggered by the car key)
    - the seat is not adjusted.
    - If need be, the currently active adjustment is aborted.
    - Additionally, the error message err_bat_low_seat_key is sent on the bus.

As convention, all key words of the running example (service names, inputs, outputs, etc.) are written in CAPITALS in the remainder of this thesis.

[1]In modern cars, on opening the car remotely the seat is adjusted according to the previously saved position.

In this chapter we presented the informal, textual requirements of our running example. In the following chapter (Chapter 3, *Notational techniques (Overview)*), we introduce notational techniques which we will make use of in the course of the methodology.

Chapter 3

Notational techniques (Overview)

In this thesis we present a service-oriented methodology for modeling usage behavior of multi-functional systems. For a pragmatic approach it is inevitable to provide suitable notational techniques. In industry, graphical descriptions for the specification and development of complex systems, provided by a number of development tools, are already in widespread use. However, many of these notational techniques lack a precise semantics. Consequently, the interpretation of its models is sometimes unclear or ambiguous.

In this chapter, we introduce the notational techniques which we suggest for our methodology. They provide the advantages of both intuitive graphical notations and a formal foundation of their semantics.

In the following section (Section 3.1, *System Structure Diagrams (SSDs)*), we explain System Structure Diagrams (SSDs). They are used to formally specify the syntactic interface and the system architecture. State Transition Diagrams (STDs, see Section 3.2, *State Transition Diagrams (STDs)*) are used to describe the semantic interface of system entities. Finally, we shortly introduce service hierarchies and service graphs. These diagrams capture the hierarchical structuring of system services and (other) relationships between services (see Section 3.3, *Service hierarchy and service graph*). For the sake of completeness, tables are mentioned in Section 3.4 (*Tables*). Tables are used to specify the static interface of services for example.

Originally, SSDs and STDs were introduced to describe the syntactic and semantic interface of *components*. However, for our methodology, we use these notional techniques for the specification of *services*. Therefore, we speak of "services" instead of "components" in the following.

Contents

3.1. System Structure Diagrams (SSDs)

In this section, we give an intuitive description of SSDs and describe their graphical representation.

We leave the formal semantics of both SSDs and STDs to Section 3.2.3 (*Semantics of STDs*). A formal definition of the semantics of SSDs in combination with State Transition Diagrams (see Section 3.2, *State Transition Diagrams (STDs)*) is given in Section 3.2.4 (*Semantics of the combination of STDs and SSDs*).

3.1.1. Intuitive description

In our methodology, we make use of System Structure Diagrams (SSDs) [Huber et al., 1997] to describe *static aspects* of a system such as its architecture. The system's architecture is viewed as a network of services exchanging data over directed channels.

We make use of non-buffering channels. This has the following reasons: First, the original AutoFOCUS semantics also makes use of non-buffering channels. Second, when using unbuffered communication, the system space is fixed during the system execution (provided fixed size data types). Consequently, model checkers in general automatically verify behavioral properties in a reasonable manner.

However, the buffering of inputs can be explicitly introduced into the specifications if needed (see Chapter 5, *Extension of basic service relationships*). Each service has a set of input channels and output channels. Data types are assigned to channels and describe the data that can be transfered over that channel.

Services can be decomposed hierarchically. A service is then hierarchically refined into sub services. The sum of the sub services has the same syntactic interface as its super service.

An SSD describes both a topological view onto a distributed system and the signature (static interface) of each service.

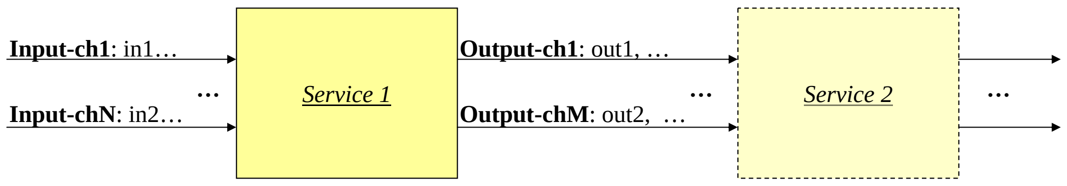

Figure 3.1.: Example for a System Structure Diagram (SSD)

3.1.2. Graphical representation

Graphically, SSDs are represented as graphs (see Figure 3.1). Rectangular vertices stand for services, directed arrows (edges) represent directed channels. Services and channels are annotated by their identifiers. Sometimes, channels are also annotated by their data types in the rest of this thesis.

In the original System Structure Diagrams, input and output ports are also introduced and depicted graphically by hollow or filled circles. In this thesis we do not make use of ports and thus omit them.

In Section 4 (*Methodology*), we will use dashed and dotted arrows to indicate channels that aim at a special purpose (see Section 4.8, *Combination of services on basis of the service relationships*). As far as semantics is concerned, these dashed and dotted channels have the same meaning as channels which are represented by solid arrows.

3.2. State Transition Diagrams (STDs)

Automata are a popular specification technique. One can distinguish between the logical model behind an automaton (state transition system) and its syntactic representation (diagram). Many different automata dialects (both state transition systems and diagram types) exist ([Maraninchi, 1991, Gurevich, 2000, Lynch et al., 2003, de Alfaro and Henzinger, 2001, Andre, 1996, Alur and Dil, 1994] - just to mention a few. See [von der Beeck, 1995] for a comparison of statecharts variants). For our approach, we choose to use AutoFOCUS State Transition Diagrams (STDs) [Huber et al., 1997] to describe the semantic interface (behavior) of services.

Analogously to the previous section, we first give an intuitive description of STDs and describe their graphical representation in the following subsections. A formal definition of the semantics of STDs and SSDs is left to Sections 3.2.3 (*Semantics of STDs*) and 3.2.4 (*Semantics of the combination of STDs and SSDs*), respectively.

3.2.1. Intuitive description

STDs are extended finite automata based on the concepts of [Grosu et al., 1996]. They are used to describe *dynamic aspects*, i. e. the behavior of a system or a service. STDs characterize the behavior by relating stimuli (i. e. inputs sent by the environment) to responses (i. e. outputs sent to the environment). Hereby, the reaction of the system depends on its current state and influences its future behavior by possibly migrating to a new state.

Timing information

In order to represent timing information and to be able to deal with priorities and interruption, STDs are endowed with a time concept. In STDs, time is represented by (discrete) time ticks. In each time tick, an input can be pending on an input channel or not. The same holds for outputs on output channels. All inputs (outputs) are read (written) simultaneously from the input (output) channels.

Furthermore, we assume that communication takes place instantaneously, i. e. no delay is introduced. This usually does not hold in reality. However, in our methodology we abstract from this problem.

State space

An STD describing a service behavior can make use of local variables. Thus, the state of a service is defined by both its control space and its data space. The control space is the current state of the finite state machine, whereas the data space is given by the current values of the local variables. In each time tick, on receiving input, the STD enters a new control state and a new data state by migrating to another state and setting the values of its local variables accordingly. Of course, also the special case that the STD stays in the same control and data space is possible.

Each STD has exactly one initial control state in which the service is at the beginning.

Transitions

So-called input and output patterns describe the transition from one control state to another. Each transition is given by a set of annotations:

- a guard {GUARD}: predicate over the data state that has to be satisfied before the transition. If the guard is not satisfied the transition is not fired but blocked as long as the predicate evaluates to false.[1]
- an input pattern INPUT: description of the message pattern that is read from the input channels
- an output pattern OUTPUT: description of the message pattern that is written on the output channels
- a post-condition {POSTCOND}: predicate over the data state satisfied after the transition

The effects of a transition are delayed by one time tick. This means the outputs of a service are visible one tick after the inputs that triggered them. This behavior is usually called "strongly causal" [Broy and Stolen, 2001].

Note that transitions in STDs can not be interrupted. However, for our methodology, the interruption (or disabling) of a service execution is very important. Thus, if the service should be interruptible between receiving an input and sending an output we have to split the transition into two transitions and an intermediary control state. We will come back on this later when introducing the methodology.

ϵ transitions, i. e. spontaneous transitions without triggering inputs, are fired in each time interval (as no evaluation of the guard and the input pattern takes place).

Hierarchical refinement of states and the history concept

Control states of STDs can be hierarchically refined. However, the hierarchy construct for STDs is only of syntactic nature and is usually used to get smaller and thus more readable diagrams. Semantically, hierarchical STDs have the same meaning as their flat correspondent.

[1]Please note, that in [Huber et al., 1997] guards are called "preconditions". However, as we want a transition to be blocked as long as the predicate is evaluated to false we prefer the term "guard".

We transfer the history concept of [Harel, 1987] to STDs. If a transition leads to the so-called history state of a hierarchically decomposed control state, the control state is entered in which the STD was when the hierarchically decomposed control state was left. If the system has not been in a sub state of the hierarchically decomposed state before, the transition leading to the history state is undefined.

Underspecification

In the following we will explain how STDs deal with underspecification.

Underspecified input patterns First we explain how the system reacts in case no input pattern is specified for an input channel within a transition. The possibilities are the following:

- Arbitrary behavior ("chaos"): In that case the reaction of the system is left open, i. e. arbitrary non-deterministic transitions are allowed.
- Ignoring of the inputs: Undefined input patterns are interpreted to result in an empty valued output and neither the control state nor the data state changes. Thus the input patterns are complete.
- Allowing for arbitrary inputs: The transition is fired no matter which input is received on the underspecified input channels.
- Buffering of the inputs: Inputs are buffered. In that case, we would have to change the semantics to buffering channels and define which input (for example the oldest input or each buffered input) would trigger a transition.

For our approach we choose the third possibility for the following reasons: First, we make use of STDs to model the behavior of services. As in our context services are partial functionalities, we also need a partial automata semantics. Second, the original semantics of STDs [Huber et al., 1997] also makes use of this semantics. Third, the chaos theory would allow for an arbitrary behavior (which would have to be refined further). However, this is too liberal for the requirements engineering phase in which the behavior of the system is to be determined. Possibility two has a total semantics and thus is inappropriate to model partial service behaviors. The fourth possibility is not in accordance with our semantics of non buffering channels (see Section 3.2.1, *Intuitive description*). We will come back to buffering in Chapter 5 (*Extension of basic service relationships*).

Hence, if no input pattern is specified for an input channel then the transition is fired independently of which value is received on this channel.

Underspecified output patterns If no output is defined for an output channel within a transition, we leave it open how the system shall react concerning this output channel. This is in accordance with our service definition which defines a service as a partial piece of functionality. Services make no statement about the system behavior for cases which are not specified.

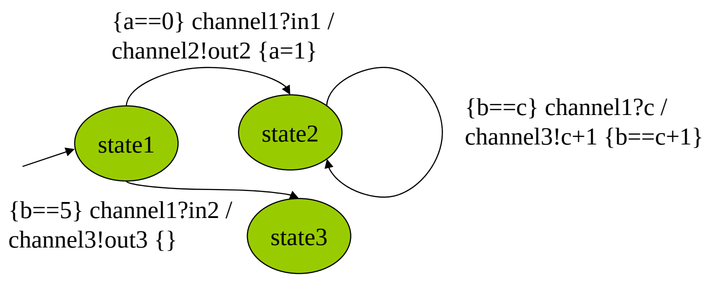

Figure 3.2.: Example for a State Transition Diagram (STD)

Underspecified assignments of variables If variables (visible to the service) are not specified in a transition, the value of these variables is unaffected by the transition, i. e. it does not change.

Underspecified transitions (summary) The combination of the different types of underspecification leads to the following result: A transition is fired no matter which value is received on underspecified input channels. It is left open how the system shall react on underspecified output channels. Furthermore, the values of underspecified local variables are not changed.

3.2.2. Graphical representation

Graphically, an STD is represented as a graph (see Figure 3.2). Labeled ovals indicate control states, labeled arrows represent transitions. Each transition is labeled by {GUARD} INPUT / OUTPUT {POSTCOND} (see Section 3.2.1, *Intuitive description*). Initial control states are indicated by an arrow with no source pointing at it.

Free variables which are local to a transition can be used to formulate more complex transitions. For example in Figure 3.2 the variable C is used to specify the behavior. If a value C is received on the input channel CHANNEL1 and C matches the value of the local variable B, then C+1 is sent (assuming that this operation is allowed for the data type of C) and B is set to C+1 (assuming that this operation is allowed for the data type of B).

For reasons of clarity, those parts of the label which are not relevant for a transition are omitted in the remainder of the thesis. For example, the channel names can be omitted if it is clear on which channel the message occurs. Then we simply write ?IN1 instead of CHANNEL1?IN1. Furthermore, if there are no guards and postcondition parts for a transition, they are not displayed either. We then just write CHANNEL1?IN1 / CHANNEL2!ACT2.

Priority concept for competing transitions

AutoFocus STDs provide a special feature - namely the *priority concept* - to deal with competing transitions. Figure 3.3 contains an example. In STATE1 it is not clear what the system has to do in case IN1 and IN2 both occur simultaneously. Which transition

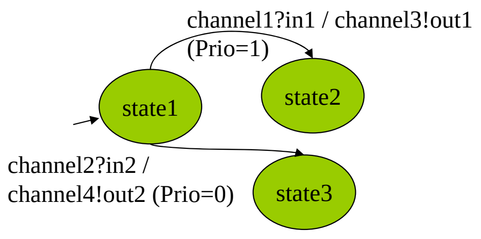

Figure 3.3.: Priority concept of State Transition Diagrams (STDs)

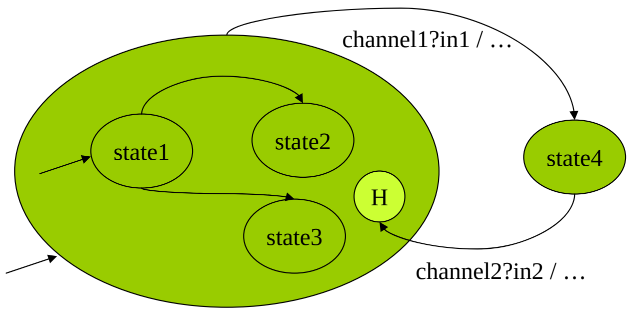

Figure 3.4.: History concept adapted for State Transition Diagrams (STD)

should be fired in that case? One convenient way is to make use of priorities. In the automaton of Figure 3.3, the transitions are given priorities to (e. g. PRIO=1). The semantics are the following: If for a state more than one transition could be fired (because the guard of more than one transition is true and the respective input actions are pending), the transition having the highest priority is fired. To that end we allow priorities from 0 to 5. If no priority is explicitly specified, we assume priority=0. Of course there might also be competing transitions having the same (and the highest) priority. The modeler has to take care of that fact.

Note that the priority concept is syntactic sugar as it also can be specified that the same behavior can be specified by more complex transitions.

History concept

For the graphical representation of the history concept see Figure 3.4. On receiving CHANNEL1?IN1, the high level state (either STATE1, STATE2, or STATE3) is left and STATE4 is entered. The transition from STATE4 to the history state of the high level state (depicted by the encircled "H") has the following meaning: When returning from STATE4 to the high level state, that state is entered which was left before. For example, if the high-level transition was fired from STATE2 to STATE4 then STATE2 is entered again.

3.2.3. Semantics of STDs

We define the semantics of STDs by means of the μ calculus logic [Park, 1976] as done in [Huber et al., 1997].[2]

The semantics of STDs are formalized by describing all possible sequences of control states. Hereby the machine which corresponds to the STD starts in the initial state. The inputs and outputs which are consumed and produced by the machine are included in the sequences. The transition relation which defines the set of possible sequences is defined by a μ calculus representation of channels, the control space, the variable state space, and the transitions.

Channels

Input and output channels are formalized by variables shared between two adjacent services. The type of the variable corresponds to the type of the channel. As a variable can only hold one value at a time (namely the value that is currently written on/read from the channel) we implicitly get the semantics of non-buffering channels.

As mentioned above, the value on a channel is restricted to a single step of the machine (a single time tick). The special value *nil* represents the absence of a message and is contained in each data type.

Variable and control state space and initial configurations

The variable state space of an STD is given by the set of all local variables and is formalized by the product of all these variables. Furthermore, a special variable is introduced for representing the current control state of the machine corresponding to the STD.

The initial configuration of the STD is given by the initial control space and the initial data space. Latter is defined by the initial variable assignments. The predicate $Init(s, x)$ on the control and variable state space formalizes the initial configuration.

Transitions

For each variable x let the primed variable x' be the value of the variable after the transition.

Let further be:

- x is the set of local variables (which are local to the service the behavior of which is given by the STD)
- $v = v_1, ..., v_k$ is a set of free variables which are local to the transition (but not local to the service)

[2]Please note that [Huber et al., 1997] also take ports into consideration for the formalization of STDs. However, as we do not use ports within STDs and SSDs we omit them here. A mapping of the STD concepts to stream processing functions can be done based on the concepts of [Fuchs and Mendler, 1994].

- $P(x)$ is a predicate[3] over x and v (the guard of the transition).
- $I(x, v) = I_1(x, v); ...; I_m(x, v)$ is a list of patterns for the input channels i_i. $I_i(x, v)$ may either be
  - empty or of the form
  - $i_i?c_i(x, v)$ with $c_i(x, v)$ being a data type constructor.
- $O(x, v) = O_1(x, v); ...; O_n(x, v)$ is a list of patterns for the output channels o_i corresponding to the input patterns.
  $O_i(x, v)$ may either be
  - empty or of the form
  - $o_i!d_i(x, v)$ with $d_i(x, v)$ being a data type constructor.
- $C(x, x', v)$ is a predicate over the local variables x and the free variables v (the post condition of the transition).

Each transition

$$S_1 \xrightarrow{P(x);I(x,v)/O(x,v);C(x,x',v)} S_2$$

is formalized by the following clause:

$$\exists v_1, ..., v_k.\ s = S_1 \wedge P(x) \wedge i_1 = c_1(x, v) \wedge ... \wedge i_m = c_m(x, v) \wedge$$

$$s' = S_2 \wedge o_1 = d_1(x, v) \wedge ... \wedge o_n = d_n(x, v) \wedge C(x, x', v)$$

For empty input patterns on channel i_i an input pattern of the form $i_i?y$ is implicitly assumed whereas y is a new transition local variable, i. e. it matches any input - even nil. The same holds for underspecified output patterns. Thus, we do not have a constructive semantics.

The complete transition relation is given by the disjunction of all clauses plus clauses for unspecified behavior. Conflicting transitions (i. e. transitions with the same start state, input patterns, and preconditions but different output patterns and postconditions) are interpreted to be solved nondeterministically. The sum of all system runs choosing one of the conflicting transitions constitutes the semantics.

Note that time is implicitly formalized. As mentioned in Section 3.2.1 (*Intuitive description*), all inputs and outputs are read simultaneously from the input and output channels, respectively. As in each time tick exactly one value can be sent on/received from a channel and the special value *nil* represents no value, time is implicitly specified.

3.2.4. Semantics of the combination of STDs and SSDs

The system is comprised of a collection of interacting services. Figure 3.5 shows the interplay of interacting services. Service C1 is hierarchically decomposed into services C1A and C1B. Thus the overall system behavior is given by the behavior of the single services (STDs) and the communication between these services (SSDs). As we assume instantaneous communication (see above), the formalization is obtained by

[3]Only prositional logic and equality are allowed.

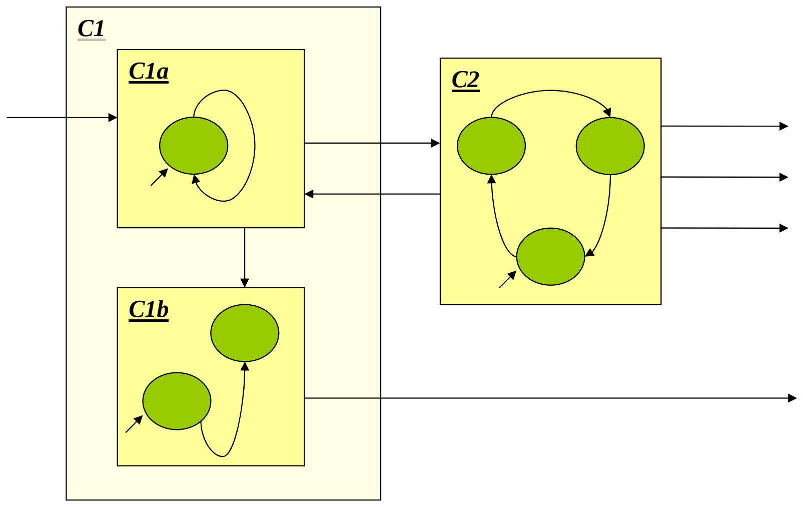

Figure 3.5.: Combination of (hierarchical) SSDs and STDs (SSD + STD)

simply combining the behaviors of the services and defining shared communication channels between services.

- Control state space: The control state space of the combined behavior is given by the product of all the control state spaces of the services.
- Variable state space: The variable state space of the combined behavior is defined by the product of all the variable state spaces of the services (renamed if needed) plus the internal channels (given by the SSD).
- Input and output channels: The input and output channels of the combined behavior are the external channels of the SSD. (The other channels are internal channels of the combined behavior.)
- Initial (control and data) states: The initial state of the combined behavior is the product of all initial states of the services.
- Transition relation: The transition relation of the combined behavior is given by the product of the transition relations of each service.

Variables can also be assigned to hierarchically decomposed services. For example, let variable z be assigned to service C1 in Figure 3.5. The question arises to which services the variable is visible. For our methodology we make us of the semantics described in [Schätz, 2007]: Local variables of hierarchically decomposed services are visible to the service itself and to all its sub services.[4] A motivation for this will be given in Section 4.8 (*Combination of services on basis of the service relationships*). Allowing more than one service to access the same variable might lead to conflicts, if two or more services want to write the same variable at the same time. We will also come back to this point in Section 4.8 (*Combination of services on basis of the service*

[4]Please note that in [Schätz, 2007], local variables are automatically made visible to all neighbor and sub services during the combination process. In order to achieve the desired semantics - that only the sub services have access to the local variable - the variable is explicitly hidden for other services. This is complementary to our procedure. We directly assign local variables to hierarchical services and declare it to be visible to all sub services.

relationships).

3.3. Service hierarchy and service graph

This thesis introduces a service-oriented methodology for modeling usage behavior of a system. The basic idea behind the approach is to combine modular service specifications on basis of their relationships. In order to visualize various kinds of service relationships we will introduce service hierarchies and service graphs in Section 4.6.2 (*Notational technique(s)*).

The hierarchical structuring of services can be graphically described by a directed graph (*service hierarchy*). Nodes of the graph depict services; parent nodes aggregate the behavior of sub services. The root of the graph contains the overall system behavior. The edges of the graph represent the hierarchical (restricted) sub service relationship between services. Service hierarchies can be enriched by horizontal relationships between services on arbitrary levels. The result is the so-called *service graph*. Vertical relationships capture dependencies between services.

Service hierarchies and service graphs are only mentioned at this place for the sake of completeness. For more information please refer to Section 4.6.2 (*Notational technique(s)*).

3.4. Tables

Additionally, we also make use of common tables to informally specify services, service dependencies, inputs, and outputs. The concrete structure of the tables is explained in the respective chapter. They are simply mentioned for the sake of completeness at this place.

3.5. Summary

In this chapter, we presented notational techniques which we make use of in our service-oriented methodology. In the following chapter (Chapter 4, *Methodology*) we introduce the methodology for modeling the black box behavior of multi-functional systems step by step.

Chapter 4

Methodology

In this chapter we introduce a methodology for modeling the black box functionality of multi-functional systems. First, basic considerations needed for the approach are made. For example the underlying system model is described. Afterward, an overview of the methodology is given. The starting point, each methodological step, and the result of the applied methodology are explained in detail in the following sections.

For each methodological step the corresponding concepts are given. Where appropriate, notational techniques are described and sub activities (of the methodological step) are presented. Furthermore, we apply the approach to the running example for each step.

At the end of this chapter, further considerations on questions related to the approach are made.

Contents

4.1. Basic considerations

In the thesis at hand, a methodology is presented which is concerned with the modeling of multi-functionality. As already mentioned in Section 1.1 (*Motivation*), so-called multi-functional systems are functionally complex systems. The complexity hereby does not arise because of complex algorithms but because of a high degree of interaction between system functionalities. In the introduction, we gave the remote unlocking of a car as an example for a multi-functional system.

In the introduction we argued that for the system class of multi-functional systems, the modeling of the pure functionality (abstracting from issues concerning distribution or technical realization) is very important to understand the complex dependencies between functionalities. Therefore we are only interested in the functionality of the system under specification.

Before we start with the description of our methodology, we make some basic considerations. First, the system model underlying our approach is sketched out. Then, the information which is contained in the informal requirements and which is to be modeled by our methodology is given a closer look at. At the end of this section, we briefly discuss the specialties of embedded systems.

4.1.1. Underlying system model

Before we explain the approach in detail, we shortly describe the underlying system model in this section.[1]

As already mentioned in the introduction, we model the system functionality from a black box perspective (usage behavior) as this is appropriate for the requirements engineering phase. For the requirements definition, the behavior of the system is looked at from an outside perspective. We model the system functionality from a *black box view*, i. e. the system behavior is only observed at the system boundary. The internal realization (*white box behavior*) is not taken into account. Especially the decomposition of functionality - which is observable at the system boundary - into sub functions and resulting function calls - which are only observable from a white-box view - are not taken into consideration.

At the system boundaries, inputs (stimuli) go into the system and outputs (system reactions) leave the system as a response to the inputs. The black box behavior of the system therefore is specified by relating input streams (of messages since the system start) and output streams (of messages since the system start) to each other. In order to structure the black box behavior, we decompose the system functionality into smaller pieces - namely services. A service is a partial function which relates inputs to outputs and thus can be seen as a projection of the overall system behavior.

The overall system functionality is established by the interplay of its services (see Figure 4.1). Services process parts of the possible inputs and produce parts of the possible outputs. (Note that inputs and outputs of different services can overlap, respectively.) Services are connected to each other by directed channels. By sending messages (so-called "actions" in our context), services can influence each other (e. g. interrupt or disable each other). However, as we model the usage behavior of the system, only messages are sent which represent *observable behavior* at the system boundaries. For example, the interruption of a service is observable from an external

[1]Note that for a comprehensive development process, the system model of the requirements engineering phase has to be refined by the system model of the design phase.

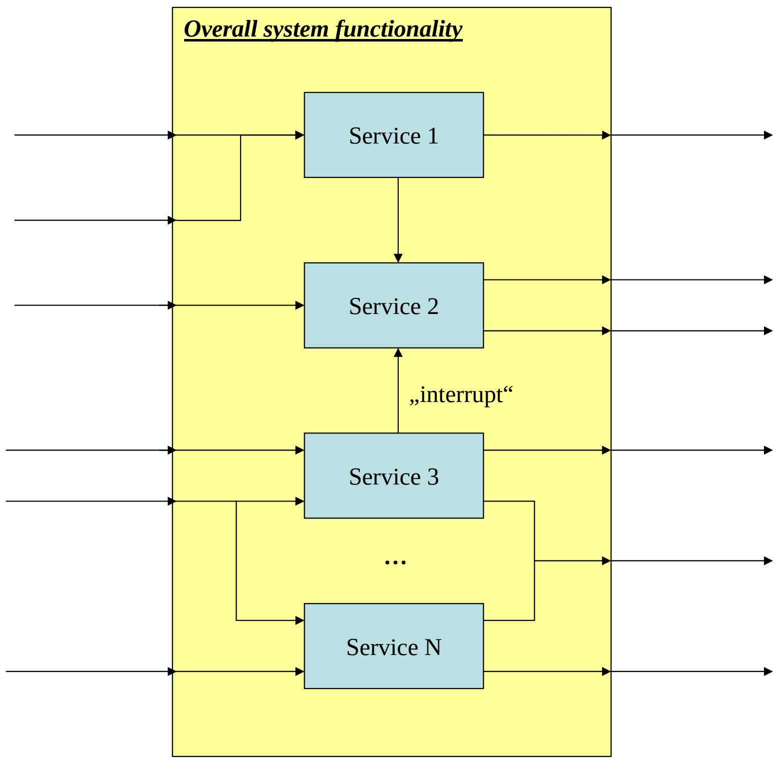

Figure 4.1.: Graphical representation of the underlying system model (SSD)

perspective and therefore is modeled. In contrast, function calls are not observable (as from an external perspective it does not matter if the functionality is realized by another function that is called or by the function itself).

Note that in Section 1.1 (*Motivation*) we claimed our methodology to be a black box approach. Strictly spoken, the approach is not a pure black box approach as it combines "smaller" services step by step to obtain the overall system functionality. Consequently, each non atomic service (see above) has an inner structure, namely the services of which it is composed. However, the aim of our methodology is to specify the usage behavior, i. e. the observable behavior of a system. We are not concerned with specifying the internal realization of the behavior. Therefore, we consider our approach to handle a black box view onto the system and call it a black box approach.

By assigning messages to channels, the channels are typed. In our methodology, we assume a global time which divides time into equidistant time intervals. The system behavior is observed between *discrete points in time*. (See Section 1.2.2, *Classification according to variation in time*.) At discrete points in time the presence or absence of input can be observed and the respective output is given out.

Furthermore, we assume that our systems are *strongly causal* [Broy and Stolen, 2001], i. e. the inputs until time interval t completely determine the output until (and inclusive) time interval t+1. Differently spoken, each system induces a time delay of at least one time unit.

Our notion of the term system is *relative*, i. e. the term system can refer to both large systems and subsystems depending on the product to be developed.

4.1.2. Constituents of functional requirements

The starting point of our methodology is a textual description of the functional requirements. In the course of the process, these informally given requirements are turned into formal models. That means that the information contained in the functional requirements is modeled (formalized). Consequently, a basic question is what the constituents of the textually given requirements are.

When having a closer look at the requirements listed in Chapter 2 (*Running example*), we can identify the following types of information:

- **Inputs and outputs**: Containing the information which are the inputs (stimuli) and outputs (reactions) of the system that can be observed at the system boundaries.
  For example, "pressing a toggle switch" is an input and "moving the back of the seat backwards" is an output.
- **Service names**: Containing the information which services the system under specification has to offer to the user.
  For example, the memory functionality via car key is a service offered to the user.
- **Services relationships**: Containing the information which relationships between services exist.
  For example, the mutual exclusion of the adjustment services is a service rela-

tionship.

- **Service behavior**: Containing the information what the behavior of the services is, respectively.
  For example, the description of how the memory functionality via car key has to look like is the description of a service behavior.
- **Persistent data**: Containing the information which persistent data a service needs for its execution.
  For example, the four dimensions of the seat position have to be saved persistently as the memory services need these data to automatically adjust the seat.
- (**Other**: Furthermore, requirements include comments like rationales. This information is omitted as it is not of relevance for the modeling of the system functionality.)

As we will see later, each step of the methodology is based on requirements of one or more categories of this classification. As we deal with the modeling of multi-functionality, the term requirement always refers to *functional* requirement in the remainder of this thesis.

Note that sometimes requirements also refer to the internal realization of the behavior. For example there may exist requirements demanding the usage of a particular algorithm. However, these functional requirements are omitted in this thesis. See also Section 7.3 (*Outlook*).

4.1.3. Specialties with embedded systems

The methodology presented in this thesis is domain independent. However, in order to illustrate our concepts we make use of a running example from the automotive domain (see Chapter 2, *Running example*). To be more precise, we make use of an embedded control system.

The functionality of the running example may seem to be of "different types", i.e some functionality is more technical than the others. For example, Requirement 10 demands that a minimum of battery power has to be available in order to use the manual adjustment functionality. In contrast, the sub requirements of Requirement 1 describe less technical but more human user oriented functionalities. These different kinds of functionalities could be named "system service" and "user service" for example. Usually the system services arise because of technical constraints and design decisions.

As some functionalities are technical the suspicion may arise that we give up the *black box perspective* onto the system behavior and look inside the system, too. However, this is not true as we will explain in the following.

Let us assume that we want to specify the functionality of the embedded system depicted in Figure 4.2 (i. e. the functionality of the sub system encircled in red). To that end we model its system functionality from a black box perspective. The embedded system is - per definition - embedded into a larger system (named "overall system" in the figure). Consequently, it has to handle input which is a consequence of a human user's input or which is input from another system. For example a user might press

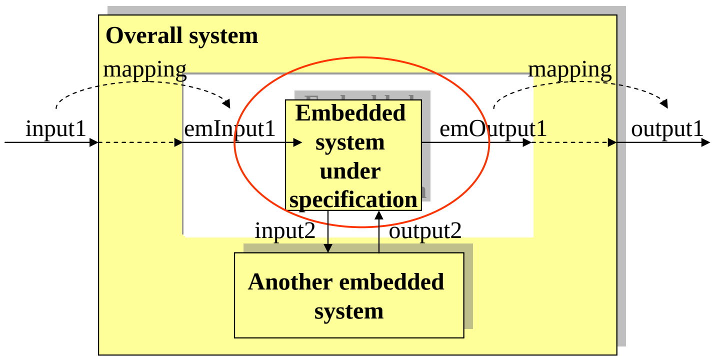

Figure 4.2.: System boundaries of an embedded system (schematic picture)

the button of the power windows in order to move the window upwards (INPUT1 in Figure 4.2). This input is transfered within the system to the responsible embedded system (e. g. per bus system). On the way to the embedded system the input might be changed into some other signal/message (EMINPUT1 in Figure 4.2) which finally arrives at the interface of the embedded system. Thus, the technical signal EMINPUT1 represents the driver's wish to open a window. This can be understood as a mapping of INPUT1 (which is an input to the overall system) to EMINPUT1 (which is an input to the embedded system).[2] However, the embedded system also communicates with other (surrounding) embedded systems. For example the energy management system may send signals to the power window control in order to stop it in case the battery is too low. The communication between embedded systems is depicted by the exchange of messages/signals INPUT2 and OUTPUT2 in Figure 4.2.

For the embedded system both kinds of functionality are of the same type as the embedded system has to relate input to output (black box view onto the functionality of the embedded system).

In an optimal development process, the question which functionality should be realized by software and which by hardware should not be answered until the design phase. (In practice however, this is not the case.) The whole functionality should be modeled independently from technical issues. For example, each car provides the possibility to mechanically unlock the car (by the door lock). The idea might arise that this functionality does not have to be formally modeled. However, the mechanical unlocking of the car can have impact on other functionalities (e. g. the alarm system). Therefore we advise to also model functionality which will later be realized by hardware.

4.2. Overview of methodological steps

In this section, we give an overview of the methodology to be presented. Each step of the methodology (including starting point and result) is explained in detail in the subsequent sections, respectively. First, we give requirements for the methodology to be presented. Based on these requirements, we will deduct the single steps of the

[2]Note, that in general, it does not have to be a 1:1 mapping between these inputs, but an n:m mapping.

methodology and describe each of it briefly.

As already mentioned, the aim of the methodology is to "transform" the textual, functional requirements of a multi-function system into formal models of the black box system functionality. The following requirements have to be fulfilled:

- **Structure of the procedure**: The methodology should consist of a structured proceeding on how to develop the formal model of the usage behavior. It should define steps and artifacts being the results of the steps.
- **Seamless transition from informal to formal specifications**: The afore mentioned gap between the informal descriptions of the functional requirements (texts) and the formal models of the system functionality should be bridged.
- **Scalability of the process**: The methodology should be applicable to systems of different scales. As the system under specification can be quite large, this problem must be handled, too.
- **Support of reuse**: Due to increasing cost pressure in industry, the methodology should strongly support reuse of most artifacts.
- **Support of distributed development**: As large-scale systems are often developed distributedly (producer supplier relationship) the methodology has to support the distributed specification of subsystems and the combination of subsystems.
- **Understanding of dependencies**: Multi-functional systems (the target systems of our approach) are characterized by a high degree of interaction and dependencies between sub functionalities. These dependencies have to be made explicit in order to understand the complex interplay.
- **Capturing of the (syntactic and semantic) system interface**: Often, the development of a system is difficult because the interface of the system has not been defined properly in the requirements engineering phase. Therefore, the methodology should explicitly describe both the syntactic and semantic interface of the system under specification.

Which consequences for the methodology can be derived from the requirements? To provide a structured procedure, the methodology has to be divided into single steps. In each step the information contained in the functional requirements has to be turned into a more formal representation. This enables a seamless transition between the informal functional requirements (starting point) and the formal model of the system functionality (result).

Modularity has to be the basic concept underlying all steps. Due to the modular development of services (which are "smaller pieces" of the system functionality), artifacts can be reused and the specification can be done distributedly.

The methodology has to provide both a local view (on single, modular services) and a global view (describing the dependencies between the single services). This reduces complexity and enables an explicit capturing of the various dependencies between services of a multi-functional system.

Figure 4.3 gives a graphical representation of the methodology to be presented in this thesis. The process can be roughly divided into two phases:

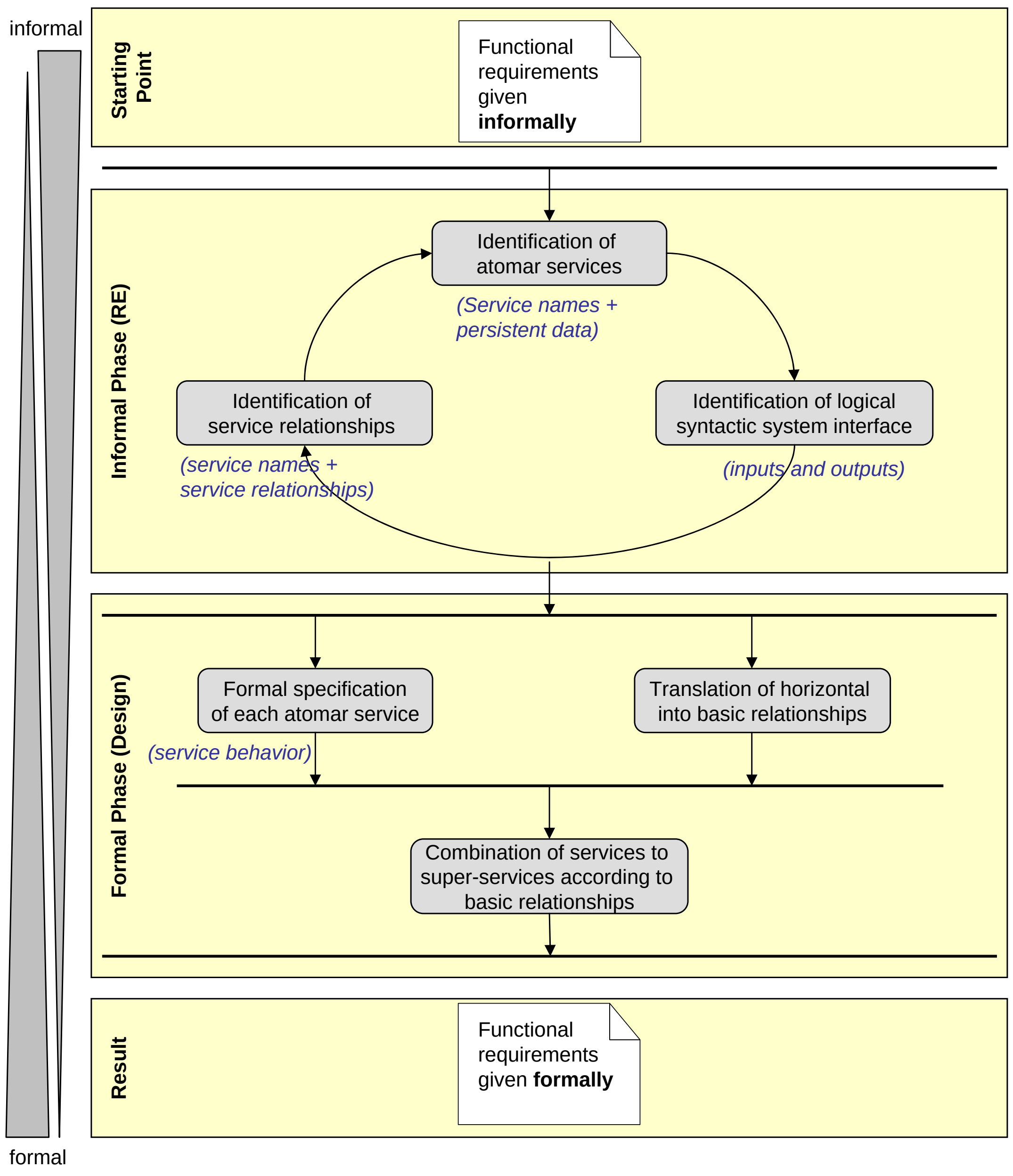
informal
Starting Point
Functional requirements given **informally**
Informal Phase (RE)
Identification of atomar services
(Service names + persistent data)
Identification of service relationships
(service names + service relationships)
Identification of logical syntactic system interface
(inputs and outputs)
Formal Phase (Design)
Formal specification of each atomar service
(service behavior)
Translation of horizontal into basic relationships
Combination of services to super-services according to basic relationships
Result
Functional requirements given **formally**
formal

Figure 4.3.: Methodology (activity diagram)

- the *informal phase* and
- the *formal phase*.

The starting point of our approach are the textual (informal) functional requirements. The *informal phase* consists of the following steps:

- **Identification of atomic services**: This step aims at identifying the "smallest" services (atomic services) which a user can access. Furthermore, data that has to be persistently saved for a correct service execution is identified. Due to reasons of complexity, the services are first - in this step - described informally (textually) and formally specified later.
  A subset of the *service names* and the *persistent data* (see Section 4.1.2, *Constituents of functional requirements*) contained in the requirements basically determine the set of atomic services and the data which have to be saved.
- **Identification of the (logical) syntactic system interface**: The result of this step is a listing of all inputs and outputs which are visible at the system boundaries and with which the system communicates with other systems. As we abstract from technical details - like signals - we speak of a *logical* syntactic system interface (see Section 4.5, *Logical syntactic system interface*).
  The *inputs* and *outputs* contained in the requirements determine the (logical) syntactic interface.
- **Identification of service relationships**: In multi-functional systems, there is - per definition - a high degree of interaction between the system functionalities (services). Service relationships capture these dependencies. In order to understand the complex interplay between services, the various service relationships between single services are captured in this step. We hereby distinguish between *vertical* and *horizontal service relationships*. Vertical service relationships define a hierarchy on services and represent an "is-contained in" (structural) relationship. *Super services* contain two or more *sub services*. Horizontal service relationships capture the dependencies between services (e. g. mutual exclusion between services). A special case of a service relationship is a *data dependency*.
  The *service names* and a subset of the *service relationships* contained in the requirements determine the horizontal and the vertical relationships.

The following steps make up the *formal phase*:

- **Formal specification of each atomic service:** In a model-based development process, the system functionality - and therefore in our case the services - have to be specified formally. In this step each atomic service is formalized. This step includes the assignment of data types to persistent data and the definition of complex data types if necessary.
  The descriptions of the *service behavior* contained in the requirements determine the models of the services, respectively.
- **Translation of horizontal relationships into basic relationships**: Arbitrary horizontal relationships between services can be thought of. As we will see later, horizontal service relationships can be reduced to a set of so-called *basic service relationships*. (As the combination of services - see next step - is done according to these basic service relationships, the horizontal service relationships

first have to be expressed on basis of these basic service relationships.)

- **Combination of services to super-services according to basic relationships**: Finally, the service specifications are combined with help of the basic service relationships. First, the atomic services are combined to more comprehensive services. (Note that our notion of service is scalable.) Then, these more comprehensive services are combined step by step until the overall system functionality is obtained. This step also includes conflict solving.

As result we obtain a formal model of the overall system functionality being comprised of modular services and their relationships.

In the subsequent sections, the starting point (textual requirements), each of the methodological steps (briefly explained above), and the result (formal model of the system behavior) are described in detail.

4.3. Starting point

Before describing the service-oriented methodology, we describe what the starting point of our approach is considered to be.

4.3.1. Concepts

As we want to model the functionality of a multi-functional system, the starting point of the approach of course are the functional requirements of the system under development. (In Section 1.2.1, *Rough outline of the approach* we already gave a definition for the term functional requirement.) Consequently, the questions arise to what extent the functional requirements have to be present at the beginning of the methodology and in what form.

As we do not primarily want to deal with issues concerning the elicitation of requirements (see Section 1.2.3, *Tasks of the requirements engineering phase*), we assume that the functional requirements are given (almost[3]) completely.

The other question to be answered is in what form the requirements are assumed to be given. One possibility could be to take current specifications (as for example given in [Houdek and Paech, 2002]) as starting point. The advantage in this case is that the approach to be presented could be used for modeling the functionality as described in existing specifications. However, current specifications are inappropriate for our approach for the following reason: Usually, specifications as can be found in industry mix up technical and logical details. For example, concrete signal names are used to describe the inputs and outputs of the system. As we propose a development process which is based on different levels of abstraction (see Section 1.1, *Motivation*) we suggest to describe the functional requirements without any hardware information. (The mapping from the abstract information to the hardware information can be done by putting the signal names in brackets or introducing an additional section in which the mapping is performed.)

[3]As a consequence of our model-based approach, missing requirements can be detected. However this is not the main focus of the approach.

Furthermore, we assume that the requirements are structured. To that end we suggest that functional requirements that belong together are grouped hierarchically. This is already done in most existing specifications. For example, the requirements belonging to the manual adjustment or the memory functionality are put in separate sections, respectively (see Chapter 2, *Running example*).

In the previous paragraphs we made assumptions in what form the functional requirements should be present at the beginning of our methodology. Of course, current specifications can also be used as the starting point for our methodology. In that case, the necessary information would have to be worked out first. For example, technical information would first have to be abstracted away. As we propose a development process along different levels of abstraction (see Section 1.1, *Motivation*), we strongly suggest that the requirements are specified appropriately right from the beginning. To that end, we will give guidelines for the informal specification of functional requirements in order to enable a preferably seamless transition to our approach in Section 4.10.3 (*Guidelines for the informal specification of functional requirements*). Of course this requires an adjustment for all people who are concerned with the system specification. Among these people are requirements engineers and designers but also customers and people of the marketing department. However, we think that this adjustment is reasonable.

4.3.2. Application to the case study

As starting point for the modeling of our running example, we make use of the requirements as given in Chapter 2 (*Running example*).

4.4. Identification of atomic services

In medium to large-scale systems, the overall system functionality can be quite comprehensive. In order to reduce the complexity, our approach is based on the following idea: The system behavior is comprised of single services which collaboratively establish the overall functionality (depending on the service relationships between them). In the first step of our methodology, we determine the so-called *atomic services* which are the "smallest" services that are visible to and can be accessed by the user. These services will be later combined to form the comprehensive system behavior (see Section 4.8, *Combination of services on basis of the service relationships*).

4.4.1. Concepts

As already mentioned in Section 1.2 (*Content of this thesis*), there exist many definitions for the term service [Meisinger and Rittmann, 2008]. Therefore, we first have to define what a *service* is in our context. Informally spoken, a service represents a piece of functionality of the system under specification. It corresponds to a use case which describes how to use a system for a specific purpose by means of interaction patterns.

A service is

- a *(partial) piece of functionality* relating system inputs to system outputs
- a *user visible* piece of functionality (a piece of functionality that can be called by a user)

Partial means, that the functionality is not specified totally. In general, the service specification leaves open how the system shall react on receiving particular inputs in particular situations. The special case of a *total* black box behavior is also a service. User visible means that the behavior is observable at the system boundaries. Services capture the interaction between the system and its environment. Note that the initiative does not necessarily lie with the user but can also lie with the system itself. With the help of services the overall system functionality can be structured into smaller pieces.

Notice the important consequence of this definition: The term service is *scalable*. Consequently, not only "small" pieces of functionality (like the adjustment of a seat in one direction) are services. Also more comprehensive functionality (like the overall power seat control functionality) can be called a service. This makes sense as we will later combine services to form bigger services (see Section 4.8, *Combination of services on basis of the service relationships*).

As the notion of service is scalable, it is difficult to say what the "smallest pieces" of functionality - i. e. the *atomic services* - are. At this place we can only give a rule of thumb. Atomic services are the "smallest" pieces of (black box) functionality that

- can be accessed/observed/distinguished by a user
- are likely to be reused[4]

Note that atomic services can be of different granularity (see Section 4.4.4, *Application to the case study*). For example both a reaction pattern (pressing a toggle switch causes the seat to move) and a more comprehensive functionality (the whole seat adjustment functionality) can be called a service. Sometimes, "events" can be considered to be modular services, too. For example, the requirement "If the velocity is too high, the adjustment of the seat has to be aborted immediately." demands for the service "too high velocity for seat adjustment". For the meantime it might not be obvious why this simple "event" should be treated as a service, too. We will come back to this point later (see Section 4.8, *Combination of services on basis of the service relationships*).

Note, that there is no 1:1 mapping between requirements and services. Usually, more than one requirement describe a single service. And sometimes a requirement can be used to identify more than one service. It requires genuine design work to identify the services out of the requirements.

[Kang et al., 2002a] suggest to analyze terminologies in order to identify services. To the authors' experience, analyzing standard terminologies is an efficient and effective procedure to identify services of a system. They also suggest to make use of service

[4]We will refine this rule of thumb later when having more information about the methodology: Our atomic services are combined to more comprehensive services on basis of the service relationships. If there exist service relationships that only refer to a sub behavior of a service, it is advisable to decompose this service into smaller sub services (atomic services). Additionally, some service relationships need the execution status of a service (active or inactive). In order to determine the execution status of a service it might be necessary to decompose the service into sub services (atomic services) as well.

(feature) categories as a service identification framework. We will come back on this idea in Section 4.6, *Combination of services on basis of the service relationships*).

In this step, we are only interested in the question which atomic services the system has to offer. The service names contained in the requirements describe these services. Therefore the atomic services are a subset of the service names contained in the requirements. In this step of the methodology we specify the atomic services *informally*. That means that no formal specification in terms of formal models is done. (The formalization of the atomic services is done later in the methodology. See Section 5.4.3, *Formal specification of modular services*.)

Some services need to operate on persistent data. For example the memory services need the previously saved position of the seat to adjust it. Therefore, we also identify this data. As services might operate on the same data[5] we introduce a repository for persistent data. In this repository, the persistent data is listed and informally described. The informal service specifications then refer to entries of this list.

For the informal specification of the services, we suggest the following constituents:

- a reference to the requirement(s) demanding the service,
- a meaningful service name,
- an abbreviation (which will later be useful when displaying the services in the graphical representations),
- a textual service description (which can be extracted from the informally given requirements and which will be used for the formalization of the service later), and
- (if needed) a reference to the persistent data needed by the service.

Note that the informal services are just listed and not structured yet. This will be done later (see Section 4.6, *Identification of service relationships*).

4.4.2. Notational technique(s)

We suggest a table to list the informal descriptions of the atomic services and the persistent data, respectively. The first table has the following columns: abbreviation (textual abbreviation), name, and description. The other table is comprised of the columns reference to requirements (containing the numbers of the requirements demanding the service), service name (textual name), abbreviation (textual abbreviation), textual service description, and reference to persistent data.

See Section 4.4.4 (*Application to the case study*) for an example for a data repository and several examples for informal specifications of atomic services.

For the service names and abbreviations, naming conventions are reasonable. For example all services describing behavior in error cases could start with "error".

[5]We will come back to this point and the problems connected to this fact in Section 4.8 (*Combination of services on basis of the service relationships*).

Table 4.1.: Specification of persistent data

Abbreviation	Name	Description
...	...	...

Table 4.2.: Specification of atomic services

Req	Service name	Abbreviation	Textual service description	Data abbr.
...	...	...	...	...

4.4.3. Methodological steps

For the first step of our methodology (the identification of atomic services), we obtain the sub activities of Figure 4.4. First, the atomic services and the persistent data are determined. Then the services and the persistent data are informally specified by tables.

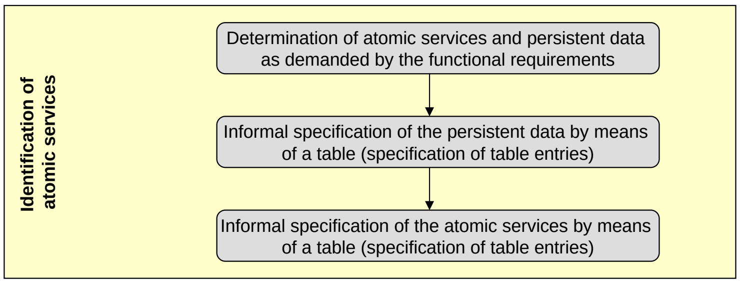
Identification of atomic services
Determination of atomic services and persistent data as demanded by the functional requirements
Informal specification of the persistent data by means of a table (specification of table entries)
Informal specification of the atomic services by means of a table (specification of table entries)

Figure 4.4.: Identification of atomic services (activity diagram)

4.4.4. Application to the case study

As far as our running example (the power seat control) is concerned, we identify the persistent data shown in Table 4.3.

Furthermore, we obtain the atomic services of Table 4.4. We go through the requirements and identify the atomic services which are visible and accessible by the user. Please note that it is not sufficient to omit the textual description of the requirement and to just give the reference number as there is not always a 1:1 mapping between requirements and services.

Remark: The question arises if it really makes sense to have such small services as ADJBACKFORW and ADJBACKBACKW. Maybe a service ADJUSTMENT OF THE BACK (ADJBACK) would be fine-grained enough. Also considering the reuse of services, the introduction of separate services ADJBACKFORW and ADJBACKBACKW does not seem to be appropriate as both services will always be reused together. However, ADJBACKFORW and ADJBACKBACKW are user visible services; the user can well distinguish between these two services. As we will see later (see Section 4.8.4, *Application to the case study*) the distinction also makes sense because of other reasons.

Table 4.3.: Specification of the persistent data of the power seat control system

Abbreviation	**Name**	**Description**
posBack	position_of_the_seat_concerning_back	...
posDist	position_of_the_seat_concerning_distance	...
posFront	position_of_the_seat_concerning_front_height	...
posRear	position_of_the_seat_concerning_rear_height	...

Table 4.4.: Specification of the atomic services of the power seat control system

Req	Service name	Abbr.	Textual description	Data abbr.
1(a)i	Adjustment of the back forwards	AdjBackForw	The back of the seat can be moved forwards by manual adjustment.	
1(a)ii	Adjustment of the back backwards	AdjBackBackw	The back of the seat can be moved backwards by manual adjustment.	
1(b)i	Adjustment of the distance increasing	AdjDistIncr	The distance of the seat can be increased by manual adjustment.	
1(b)ii	Adjustment of the distance decreasing	AdjDistDecr	The distance of the seat can be decreased by manual adjustment.	
1(c)i	Adjustment of the rear area up	AdjRearUp	The height of the rear area of the seat can be increased by manual adjustment.	
1(c)ii	Adjustment of the rear area down	AdjRearDown	The height of the rear area of the seat can be decreased by manual adjustment.	
1(d)i	Adjustment of the front area up	AdjFrontUp	The height of the front area of the seat can be increased by manual adjustment.	
1(d)ii	Adjustment of the front area down	AdjFrontDown	The height of the front area of the seat can be decreased by manual adjustment.	
5	Front door open	FrontDoorOpen	The manual adjustment of the seat is only possible if the front door is opened.	
10	Err low battery manual	ErrLowBatMan	In case the battery power is too low, the manual adjustment is not performed.	
11	Err high velocity	ErrHighVelo	The memory functionality can only be executed if the car velocity is not too high.	
12	Memory functionality via button	MemoryButton	The memory functionality can be triggered by pressing the button attached inside the driver's door. (...)	posBack, posDist, posFront, posRear
12	Memory functionality via car key	MemoryCarKey	The memory functionality can be triggered by pressing the respective button of the car key. (...)	posBack, posDist, posFront, posRear
14	Err low battery button	ErrLowBatButton	In case the battery power is too low (lower than 10V) to perform the memory functionality, the seat is not adjusted.	
16	Err low battery car key	ErrLowBatKey	In case the battery power is too low (lower than 10V) to perform the memory functionality, the seat is not adjusted.	

4.5. Identification of the logical syntactic system interface

A system communicates with users. The term user refers to both human users and other systems (e. g. if an embedded system is specified). It is important to determine the system boundaries (system interface), i. e. which inputs (stimuli) go into a system and which outputs (response) come out of a system. Often, during requirements engineering the explicit specification of the system boundaries is neglected which results in problems during later phases of the development process.

The interface can be distinguished into the *syntactic* and the *semantic interface*. The syntactic interfaces describes which inputs (in general) go into the system and which outputs (in general) come out of a system. The semantic interface relates the inputs to outputs.

In this step of the methodology, we specify the syntactic interface of the overall system. That means that the inputs and outputs due to which the system communicates with its environment are specified but not related to each other. The semantic interface is determined later (when specifying the behavior of the services).

4.5.1. Concepts

We look at our system from a logical view where no technical details are involved. Therefore, we do not speak of messages or signals that are inputs for the system. We are rather interested in the *meaning* of the inputs and outputs from a logical point of view. We are not interested in how the inputs are later technically realized (e. g. by signals or messages). For example, we do not say that VEH_SP is the signal or the message that goes into the system, but that a signal or message representing the VEHICLE_SPEED is the input. We thus speak of *input actions*. Analogously, we abstract from technical information when referring to outputs and speak of *output actions*. For example the output action WINDOW_MOTOR_DOWN stands for a signal or message causing the motor of the window to go down.

The input and output actions of a system together determine the so-called *logical syntactic interface* of the system. Although our approach only models the system functionality on a logical level (and does not take into consideration more concrete abstraction levels) we nevertheless explicitly speak of *logical* syntactical interfaces to sharpen the reader's understanding of different levels of abstraction. The term *logical action* refers to the sum of both *input actions* and *output actions*.

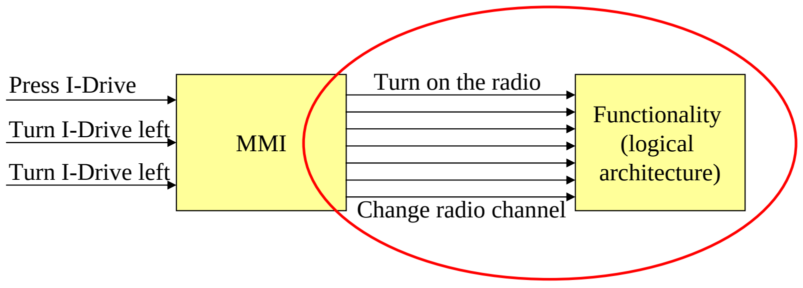

Figure 4.5.: Logical input and output actions - abstraction of the MMI (SSD)

Besides abstracting from technical issues, the logical actions also abstract from aspects concerning the man machine interface (MMI). For example, in modern BMWs an I-Drive is used to navigate through the menu. By pressing or turning the I-Drive to the left and the right, different menu items (e. g. turn on the radio, change radio channel) can be chosen. However, for the determination of the logical actions it is unimportant how the action is caused but only that it is caused. This has to be considered when determining the logical actions. Figure 4.5 illustrates this fact. This abstraction from the MMI can also be considered to be a shift of the system boundaries.

The logical input and output actions can be extracted from the informally given requirements. The logical inputs are the stimuli with which the user communicates with the system. For example, the wish to move the back backwards. The logical outputs are the response of the system, e. g. the commands for the seat motors to actually move the back.

As already mentioned in Section 4.1.1 (*Underlying system model*), the system behavior is established by a set of communicating services. By exchanging logical actions, services can influence (e. g. interrupt) each other. To that end, services are connected by directed, so-called *logical channels*. Depending on the perspective of a service, we face *logical input channels* and *logical output channels*. These logical channels will be mapped to concrete technical channels (e. g. between a control unit and a bus) later in the development process.

For our methodology, we introduce a logical channel for each input action and each output action, respectively. The motivation for this is the following: A service that is connected to an input channel receives all of the input actions that are sent on this channel. As a service - per definition - is a partial behavior it does not necessarily have to handle each of the actions. However, as we do not want to make the service specification more complex than needed, we only add those input and output actions to a service interface which are actually needed by the service. Later, when the logical service architecture is mapped to a concrete component architecture, several input messages can be sent on a concrete channel as components - per definition - are total behaviors. A logical channel is automatically typed when the logical action is assigned to it.

The question arises what should be encoded in a logical (input or output) action. In order to answer this question we have a look at the technical counterparts from which the logical actions shall abstract. We obtain the following classes:

- **Concrete values**: Logical actions abstract from concrete values being transmitted by signals or messages. For example, sensor values like DOOR_OPEN.
- **Exceeding of a value**: Logical actions can stand for the exceeding of a certain value. For example, SPEED_TOO_HIGH would represent all speed values greater than 45km/h (depending on the context).
- **Exceeding of a set of values**: Logical actions may represent the exceeding of a set of values. For example, the logical action SPEED_CHANGES stands for the change of the current speed value.
- **Sets of values**: A logical action can be the abstraction of a set of values. For example, the logical action MEDIUM_SPEED could represent a technical speed

value between 20 and 50km/h.

Theoretically speaking, it does not matter from which technical signals a logical action abstracts. The important thing is that the mapping between logical actions and their technical realization is correct. Please note that this mapping is not trivial but in fact very complex and has to be investigated in detail for a development process along different levels of abstraction. However, one desired application of the result of our methodology (the formal model of the usage behavior of the system) is to verify it. To that end, the dependencies between the actions have to be known. For example, the logical input actions DOOR_OPEN and DOOR_CLOSED exclude each other. Therefore this situation that both actions occur at the same time does not have to be specified. Thus, the dependencies between the actions should be captured. Please note that only dependencies of the logical level should be captured in the first iteration of the development process. Dependencies which are specific to the technical realization should not be taken into account when the functionality is determined in the first instance. In a second iteration - after the technical realization has been defined - also dependencies which are a result of design decisions can be considered in order to reduce the complexity of the model. An example for that is a toggle switch which automatically ensures that the signals (one per position of the switch) can not be sent simultaneously.

Capturing dependencies between logical actions (like ACTION1 and ACTION2 exclude each other, ACTION1 is a negation of ACTION2, the change in value of ACTION1 is ACTION2, etc.) is an interesting topic and definitely needed for verifying the formal model of the system functionality. However, we do not consider it in this thesis and leave it for future research.

For each (input or output) action, an intuitive and unique name and a short description has to be given. Additionally, it has to be specified on which logical channel the action can be sent. A logical action can only be assigned to one channel (the channel that was introduced for this action) and one channel can only transmit one logical action which is visible at the system boundaries. As there is a 1:1 mapping between logical actions and logical channels the question arises why we need to explicitly name logical channels. The reason is the following: Later in the methodology we will also introduce *internal* logical channels between services in order to realize the dependencies. In order to influence other services, services can send (instances of) the same logical action to several services.[6] The names of the logical channels are then needed to identify the right channel. Although logical channels that carry logical actions which are visible at the system boundaries (i. e. logical actions which are specified in this step of the methodology) do not need to have a name, we also name them to provide a unified procedure. Otherwise we would have to distinguish between named and unnamed logical channels which would of course also be possible.

Notice that we do not specify for each (input or output) action from what user/service or to what user/service it is sent. This increases reuse as the system under development could be designed for another environment or the environment could change. No changes to the logical syntactic interface would be necessary in that case. Moreover, references to requirements in which the actions are used have to

[6]Furthermore, as far as *internal* logical channels are concerned, a channel can also carry more than one logical action.

be given.

The input actions can be determined by going through the requirements and identifying the stimuli that trigger the service.

As already mentioned in the definition for the term functional requirement (see Section 1.2.1, *Rough outline of the approach*), functional requirements may also explicitly state what the system should not do. In that case we just negate an output action, e. g. NOT SEAT MOTOR BACK FORWARDS.

Please note that the range of logical actions is arbitrary. In some cases a two-valued range might be appropriate, in other cases the range is larger. For example the input action VELOCITY TOO HIGH could be Boolean. VELOCITY TOO HIGH == true would then mean that the action can be observed at the system boundary (and thus the speed of the car is too high for adjusting the seat); VELOCITY TOO HIGH == false would then mean the opposite (namely that the speed of the car is not too high). The information about the current seat position could be encoded by an integer variable of an appropriate data type, for example. For each logical action its data type has to be specified.

Note that some logical (input and/or output) actions might not be known from the beginning. Sometimes they are not identified until the formal specification of the services (see Section 5.4.3, *Formal specification of modular services*). In that case, the actions have to be added later to the table(s).

4.5.2. Notational technique(s)

For the description of the logical actions (and their channels), we suggest tables. The columns of the tables are: reference to requirement(s), name of action, abbreviation, data type, description, and logical channel (on which the action can occur). Input (output) actions may only occur on input (output) channels. In the description column, the meaning of the assignments of the actions have to be explained (according to the data type). Please note that the sources of input actions (targets of output actions) are not specified as this enhances reuse. The name and the description of an action are enough to identify it and to understand its meaning.

Table 4.5.: Specification of logical (input and output) actions

Reqs	Name of action	Abbreviation	Data type	Description	Channel
...	...	...	...	...	

In order to visualize the logical syntactic interface (logical channels and logical actions) graphically, we suggest System Structure Diagrams (SSDs, see Section 3.1, *System Structure Diagrams (SSDs)*). Figure 4.6 contains an exemplary SSD with the input channels a, b, and c and the output channels d and e, respectively. Input channel a can transmit the input action INACT1, for example.

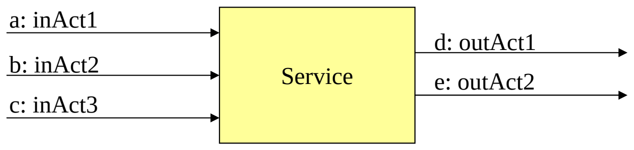

Figure 4.6.: (Graphical) Specification of the logical syntactic interface (SSD)

4.5.3. Methodological steps

This step of the methodology can be divided into the sub activities show in Figure 4.7. We go through the requirements and identify the logical input and output actions and assign them to logical input and output channels, respectively. Each input (output) action is assigned to one input (output) channel and vice versa. We specify both logical (input and output) actions and (input and output) channels in the respective tables and by means of an SSD.

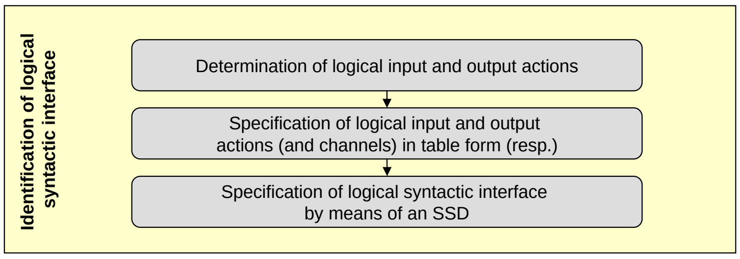
Identification of logical syntactic interface
Determination of logical input and output actions
Specification of logical input and output actions (and channels) in table form (resp.)
Specification of logical syntactic interface by means of an SSD

Figure 4.7.: Identification of the logical syntactic system interface (activity diagram)

4.5.4. Application to the case study

For our running example, we obtain the logical input and output actions shown in Tables 4.6 and 4.7. (Due to limitations of space we omit the names of the logical input and output channels here. For each action a channel with the same name or CHANNEL$< name >$ could be introduced for example.)

Table 4.6.: Input actions of the power seat control system

Reqs	Name	Abbr.	Data type	Description	I-Channel
1(a)ii	adjust back backwards	back_backw	Bool	The user wants to move the back backwards by manual adjustment. (1 means that the command is currently pending; 0 means that the command is not pending.)	
1(a)i	adjust back forwards	back_forw	Bool	The user wants to move the back forwards by manual adjustment. (1 means that the command is currently pending; 0 means that the command is not pending.)	
1(b)i	adjust distance increasing	dist_incr	Bool	The user wants to increase the distance of the seat by manual adjustment. (1 means that the command is currently pending; 0 means that the command is not pending.)	
1(b)ii	adjust distance decreasing	dist_decr	Bool	The user wants to decrease the distance of the seat by manual adjustment. (1 means that the command is currently pending; 0 means that the command is not pending.)	
1(c)i	adjust rear area up	rear_up	Bool	The user wants to increase the height of the rear area of the seat by manual adjustment. (1 means that the command is currently pending; 0 means that the command is not pending.)	
1(c)ii	adjust rear area down	rear_down	Bool	The user wants to decrease the height of the rear area of the seat by manual adjustment. (1 means that the command is currently pending; 0 means that the command is not pending.)	
1(d)i	adjust front area up	front_up	Bool	The user wants to increase the height of the front area of the seat by manual adjustment. (1 means that the command is currently pending; 0 means that the command is not pending.)	
1(d)ii	adjust front area down	front_down	Bool	The user wants to decrease the height of the front area of the seat by manual adjustment. (1 means that the command is currently pending; 0 means that the command is not pending.)	

(Table continued on the next page.)

(Table of previous page continued.)

Reqs	Name	Abbr.	Data type	Description	I-Channel
(see later)[7]	position sensor back	pos_back	Integer[1..21]	This action represents the current position of the back of the seat. (10 is the middle position; 1 is the position with the back most forwards, 21 is the position in which the back is put most backwards)	
(see later)	position sensor distance	pos_dist	Integer[1..21]	This action represents the current position of the seat concerning the distance. (...)	
(see later)	position sensor rear	pos_rear	Integer[1..21]	This action represents the current position of the rear area of the seat. (...)	
(see later)	position sensor front	pos_front	Integer[1..21]	This action represents the current position of the front area of the seat. (...)	
3	adjust by button	adjust_by_button	Bool	The driver wants to adjust the seat by pressing the memory button (memory functionality).	
3	adjust by car key	adjust_by_car_key	Bool	The driver wants to adjust the seat by pressing the button of the car key (memory functionality).	
5	front door is opened	front_door_opened	Bool	This action represents the information if the front door is opened. (1 means that the front door is currently opened; 0 means that the front door is currently closed.	
10	battery too low for manual adjustment	low_battery_for_manual	Bool	This action contains the information if the battery power is too low in order to allow for manual adjustment. (1 means that the battery power is too low; 0 means that the battery power is not too low.)	
11	velocity too high for memory functionality	velocity_too_high	Bool	This action contains the information if the car velocity is too high to allow the usage of the memory functionality	
14	battery too low for the memory functionality via seat button	low_battery_for_button	Bool	This action contains the information if the battery is too low in order to allow for memory functionality triggered by the button. (1 means that the battery is too low; 0 means that the battery is not too low.)	

(Table continued on the next page.)

[7]The introduction of the input actions sent by the position sensors will be justified later (see Section 4.7.4, *Application to the case study*).

(Table of previous page continued.)

Reqs	Name	Abbr.	Data type	Description	I-Channel
16	battery too low for the memory functionality via car key	low_battery _for_carkey	Bool	This action contains the information if the battery is too low in order to allow for the memory functionality triggered by the car key. (1 means that the battery is too low; 0 means that the battery is not too low.)	

Table 4.7.: Output actions of the power seat control system

Reqs	Name	Abbr.	Data type	Description	O-Channel
2	motor back forwards	m_back_forw	Bool	Instructs the motor to move the back forwards (1 inch, for example). (1 means that the command is currently pending; 0 means that the command is not pending.)	
2	motor back backwards	m_back_backw	Bool	Instructs the motor to move the back backwards (1 inch, for example). (1 means that the command is currently pending; 0 means that the command is not pending.)	
2	motor distance increasing	m_dist_incr	Bool	Instructs the motor to move the seat backwards (1 inch, for example). (1 means that the command is currently pending; 0 means that the command is not pending.)	
2	motor distance decreasing	m_dist_decr	Bool	Instructs the motor to move the seat forwards (1 inch, for example). (1 means that the command is currently pending; 0 means that the command is not pending.)	
2	motor rear area up	m_rear_up	Bool	Instructs the motor to move the rear area down (1 inch, for example). (1 means that the command is currently pending; 0 means that the command is not pending.)	
2	motor rear area down	m_rear_down	Bool	Instructs the motor to move the rear area down (1 inch, for example). (1 means that the command is currently pending; 0 means that the command is not pending.)	
2	motor front area up	m_front_up	Bool	Instructs the motor to move the front area up (1 inch, for example). (1 means that the command is currently pending; 0 means that the command is not pending.)	
2	motor front area down	m_front_down	Bool	Instructs the motor to move the front area down (1 inch, for example). (1 means that the command is currently pending; 0 means that the command is not pending.)	
10	error battery too low for man adj	err_bat_low_seat_man	Bool	Error message in case the battery power is too low to perform the manual adjustment.	
14	error battery too low for mem adj button	err_bat_low_seat_button	Bool	Error message in case the battery power is too low to perform the adjustment via memory (called by the seat button).	
16	error battery too low for mem adj car key	err_bat_low_seat_key	Bool	Error message in case the battery power is too low to perform the adjustment via memory (called by the car key).	

Figure 4.8 represents the logical syntactic interface graphically.

4.6. Identification of service relationships

So far, the single, atomic services have been captured in a modular fashion (separately) but not related to each other. In this section we will relate services.

4.6.1. Concepts

As mentioned in the introduction, multi-functional systems are characterized by a high degree of dependencies between functionalities. These dependencies have to be captured and made explicit. Only then, the "big picture" about the system behavior can be sketched out. In order to relate services to each other we have to identify service relationships. A service relationship characterizes the effects that a service has on (the modular specification of) another service. For example Requirement 8 demands that for each adjustment only one direction of movement (e. g. up or down) can be carried out at once.

As from our perspective the system is a black box behavior, only relationships that are visible at the system boundaries are taken into consideration, i. e. only relationships between services that a user can observe are captured. This is appropriate for the requirements engineering phase which describes the observable system behavior. Other relationships, e. g. call or trigger relationships, which can only be detected inside the system are appropriate for the design phase. The latter focuses on a decomposition of the black box functionality (usage behavior) into sub-functions and function calls. For the remainder of this thesis, the term "service relationships" always refers to those relationships which are observable at the system boundaries (and not only observable from a white-box perspective on the system functionality).

When taking a closer look at the requirements of our running example, we see that some requirements do not refer to single services, but to a set of services. For example Requirement 10 demands that each manual adjustment (BACK FORWARDS, BACK BACKWARDS, INCREASE DISTANCE, DECREASE DISTANCE, REAR AREA UP, REAR AREA DOWN, FRONT AREA UP, FRONT AREA DOWN) is not possible (or if need be has to be aborted) if the battery power is too low. This is just an "abbreviation" of demanding the disabling of each of the single services. In a case like that, we introduce a super service (in this example MANUAL ADJUSTMENT) aggregating sub services (in this example BACK FORWARDS, BACK BACKWARDS, INCREASE DISTANCE, DECREASE DISTANCE, REAR AREA UP, REAR AREA DOWN, FRONT AREA UP, FRONT AREA DOWN). Furthermore, we introduce a horizontal relationship between the battery service and the super service (instead of introducing relationships between the battery service and each of the single services, respectively).

Based on the thoughts outlined in the previous paragraphs, we make the following explicit distinction into:

- vertical (service) relationships and
- horizontal (service) relationships

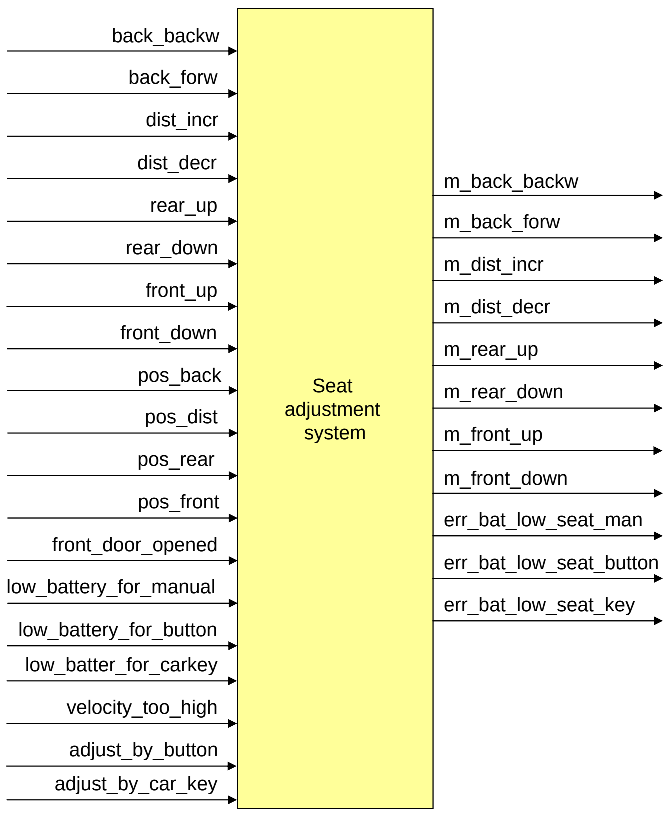

Figure 4.8.: Logical syntactic interface of the power seat control system (SSD)

Vertical service relationships

Vertical relationships between services introduce a hierarchy of services. This hierarchy can be (graphically) represented by a directed graph which we call *service hierarchy* (see Figure 4.9).[8] The parent node of a service represents the so-called *super service* which aggregates services. The children of a node represent the so-called *sub services* which are aggregated by the super service.

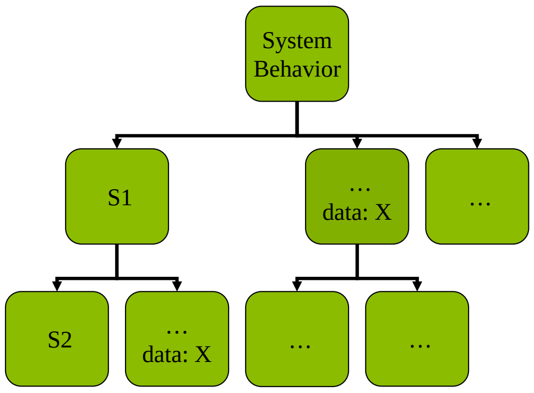

Figure 4.9.: Service hierarchy (directed graph)

The edges of the service tree (vertical relationships) represent the *restricted sub service relationship* as formally founded in [Broy, 2007]. The restricted sub service relation is a more general case of the sub service relation. Informally spoken, a service S1 is a *sub service* of another service S2, if the behavior of S1 is "always" contained in the (more comprehensive) behavior of S2. S2 thus is a projection of the behavior of S1. The super service "combines" the behaviors of its sub services. For example, the services ERR LOW BATTERY BUTTON and ERR LOW BATTERY CAR KEY are sub services of the service ADJUSTMENT BY MEMORY. They behave according to the modular specification also in the comprehensive behavior of the super service.

If the service S2 is a *restricted sub service* of the service S1, S2 is only a sub service of S1 for specific input from a subset of all possible inputs. The service ADJUSTMENT OF BACK is a restricted sub service of the service MANUAL ADJUSTMENT for example. Due to Requirement 9 (demanding that at most two directions can be carried out at once) the service ADJUSTMENT OF BACK might not be performed although it is triggered manually by the driver. This is true for the situations (system states) in which already two other adjustments are currently being carried out. Therefore, the service is only a restricted sub service (and not a sub service).

In [Broy, 2007] a super service contains the behaviors of its sub services. That means that it can have additional behavior (which is not defined by sub services), too. In the thesis at hand, a super service is the "minimal" behavior that contains the behaviors of its sub services and realizes the horizontal service relationships (see below). The root of the service hierarchy contains the behavior of all services and therefore contains the overall system functionality.

[8] Although the notational techniques for this step of the methodology are introduced in 4.6.2 (*Notational technique(s)*), we already give a preview at this place in order to explain our concepts.

When having a closer look at the services of our running example (and other case studies), we see that almost always services are related by the restricted sub service relationship and not by the sub service relationship. Additionally, we figure out that the horizontal relationships between services determine if a service is a sub service or a restricted sub service of another service. For example considering again the example of the previous paragraph. The horizontal relationship "at most two directions can be carried out simultaneously" is responsible for the restricted sub service relationship between the services ADJUSTMENT OF BACK and MANUAL ADJUSTMENT. Therefore, in the thesis at hand we do not explicitly distinguish between the sub service relationship and the restricted sub service relationship. In the remainder we will use the term "sub service relationship" although we refer to the restricted sub service relationship as formally defined in [Broy, 2007].

The leaves of the tree are the *atomic services* (see Section 4.4, *Identification of atomic services*). The inner services are more comprehensive behaviors combining sub services. In the remainder of this thesis we call the services which are represented by inner nodes of the service hierarchy *combined service*s. The combined services are all services except the atomic services. The overall system functionality - which is represented by the root of the service hierarchy - is the "most comprehensive" combined service. The introduction of a super service only makes sense, if the super service aggregates at least two services.

The names of the super services are often explicitly given in the requirements. Sometimes they are only given implicitly. In that case, a service name has to be introduced.

Horizontal service relationships

Horizontal service relationships are all other relationships except the sub service relationship. As mentioned above the relationship XOR is an example for a horizontal relationship between services. Other examples are DISABLE, INTERRUPT, and MAX2INPARALLEL (the latter demands that among a set of services at most two services can be executed in parallel). The horizontal relationships can be of an arbitrary arity, i. e. they can exist between two, three, or n services depending on the relationship. The service hierarchy is enriched by the information of the horizontal relationships. The service hierarchy *and* the horizontal relationships are captured in the service graph (see Figure 4.10).[9] The arrows are labeled by the names of the service relationships, respectively. (The meaning of the PARAMETER is explained below.).

Horizontal relationships may exist between arbitrary services. There may also exist horizontal service relationships between services of different levels. However, this does not mean that a tree is badly structured and has to be restructured. After Section 4.8 (*Combination of services on basis of the service relationships*) it will be obvious that our methodology can also handle these cases.[10]

[9]Again we give a preview of the notational techniques used in this step of the methodology

[10]For the requirements engineering phase the hierarchical structuring of usage behavior as suggested above seems most appropriate. For the subsequent design phase however, other structuring forms might be better. In the design phase the usage functionality is broken down into smaller system functions which are not (necessarily) observable at the system boundaries. This will most probably have influence on the structuring of the functionality. Due to limitation of scope, these questions are not investigated in this thesis.

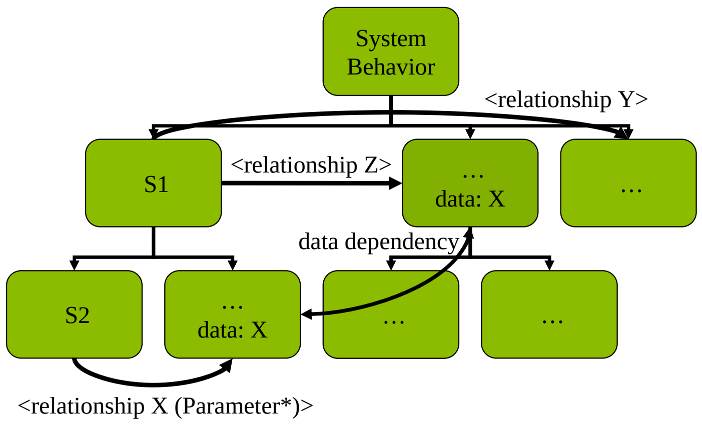

Figure 4.10.: Service graph (directed graph)

The super services are a sub set of the service names in the requirements. The atomic services and the super services together form the set of all services. The service relationships can be extracted from the requirements (service relationships).

Basic service relationships In the following paragraphs we will have a closer look at the horizontal service relationships. We first define a set of *basic* (primitive, simple) service relationships and show how the horizontal service relationships can be put down to these basic relationships.

In our methodology, we look at the system behavior from a black box perspective. Therefore, we first have to identify the "simplest" effects that a service can have on another service. These "simplest effects" are then represented by basic service relationships. We obtain the following *basic service relationships*:[11]:

A service S1 can influence a service S2 in the following "basic" ways:

- ENABLE: If S1 enables S2, S1 puts S2 in a state in which S2 can be executed. This does not mean that S2 is actually executed, but that it is *possible* to execute S2.
- DISABLE: If S1 disables S2, S1 puts S2 in a state in which S2 can not be executed. If the service is currently being executed, it is stopped and can not be executed.
- INTERRUPT: If S1 interrupts a service S2, the execution of S2 is interrupted.
- CONTINUE: If S1 continues a service S2, S2 is continued at the same point of the service execution where it was interrupted before.
- RESET S1: If S1 is reset, its execution is stopped and S1 is put in its initial state. (When called the next time, the service S1 will start right from the beginning again.)
- INDEPENDENT: If S1 does not have any influence on the behavior of service S2, we call S2 independent of S1.

Semantically, DISABLE and INTERRUPT are equivalent as both prevent a service from

[11]In Chapter 5, *Extension of basic service relationships* we will discuss this set of basic service relationships again and enlarge it.

being executed or stop its execution if need be. However, to continue a service (see basic service relationship CONTINUE) it is necessary to know at what place of the execution it was stopped. Therefore, we distinguish between DISABLE and INTERRUPT.

Complex horizontal service relationships Based on these basic relationships, *complex horizontal service relationship* can be defined. Complex horizontal service relationships are all horizontal service relationships which are not basic ones. For each horizontal relationship a stereotype with the name of the service relationship (or an abbreviation) has to be introduced. Complex horizontal service relationships can be put down to basic service relationships. For example an XOR relationship, demanding that two services are not executed in parallel, can be put down to the ENABLE and DISABLE service relationships. Consider again Requirement 8 which demands that for each adjustment only one direction of movement (e. g. rear area up or down) can be carried out at once. Assuming that the service which was called first is to be executed, we obtain the following setting: In case the upwards direction is currently performed (the service ADJUSTMENT OF THE REAR AREA UP is carried out), the service ADJUSTMENT OF THE REAR AREA DOWN is disabled (and vice versa). After the service ADJUSTMENT OF THE REAR AREA UP has finished, the service ADJUSTMENT OF THE REAR AREA DOWN is enabled again. Note, that most complex relationships are based on the execution status of a service. For example for disabling and enabling the service ADJUSTMENT OF THE REAR AREA DOWN, we have to know if the other service being executed. We will come back to this point in Section 4.8 (*Combination of services on basis of the service relationships*).

Some of the complex horizontal service relationships need additional information. For example, the XOR relationship of the example above does not say anything about what to do if a service S2 is called while another service S1 is being executed. Shall S1 be aborted in favor of S2? Or shall S1 be finished and S2 not started? Shall one of the services (S1 or S2) have a higher priority concerning the right of execution? Consequently, the XOR relationship has to be enriched with further information. At this point we face two possibilities: We can either introduce refinements of the XOR relationship or introduce a parameter for it. We take the second possibility (see Section 4.6.4, *Application to the case study*). For example, the XOR relationships is further specified by a parameter PRIORITY. The codomain of this parameter is {RUNNING, STARTED, <SERVICE>}. If the parameter is set to RUNNING, the service currently being executed is not stopped. If it is set to STARTED, the currently running service is stopped in favor of the new service. <SERVICE> stands for one of the services, to which the relationships refers meaning that this service always has the highest priority.

By introducing parameters to service relationships we obtain a nice side effect: missing requirements can be detected. For example when taking a closer look at the requirements of our running example again (see Chapter 2, *Running example*), we see that no information is given on which value to assign to the parameters of the XOR relationship. Although the identification and elicitation of (missing) requirements is not in the scope of this thesis (see Section 1.2.3, *Tasks of the requirements engineering phase*), it is partly covered by our methodology.

Note that horizontal service relationships can be symmetric or asymmetric and - as mentioned above - of arbitrary arity.

No service relationship between services implicitly represents the INDEPENDENT relationship. We therefore will not explicitly use this relationship in the remainder of the thesis. It might be wanted to explicitly specify if two services should be independent. Thus, the explicit use of the INDEPENDENT service relationship would be wanted. However, this enhances the complexity of the model. Therefore we decided to not model independence explicitly.

Data dependencies

As mentioned above, services might operate on the same persistent data. Data dependencies are also relationships between services which have to be captured. Although at this stage of the methodology we are still informal, we have to take a look how persistent data will be realized later during the formalization process. Only then, we can prepare the information in a target-oriented way.

Persistent data will later be realized by variables (see Section 4.7.1, *Concepts*). The question arises what the scoping of these variables has to be like. We face three possibilities:

- Global variables
- Variables local to atomic services
- Variables local to sub services of a service

Global variables would be visible to all services. Consequently, each service could read and write the persistent data. This is problematic for the following reasons: First, conflicts might occur when more than one service want to write the variable at the same time. Implicit communication is possible due to the shared variables. The concept of information hiding is not complied with. Thus, global variables couple services too tightly.

If persistent data was local to an atomic service, services would have to exchange data in order to operate on the same data. This could either happen in a push or pull manner. In the first case, a service would sent the data to all subscribing services in case it modified the data. In the second case, a service needing the data would explicitly request it from the service which the data is local to. However, from a requirements engineering perspective this is not appropriate as the sending of data can not be observed from the usage view.

The third possibility assigns data to a hierarchically decomposed service. In this case the data is visible to all sub services. Although the data is not visible within the overall system, the problems mentioned for global variables still hold for the set of sub services.

With all the three alternatives we face a basic complex of problems. We choose possibility three as it is most appropriate for our approach. If services operate on the same persistent data, the variable (representing the persistent data) is assigned to their least common parent. Consequently, all sub services have access to the variable. This might be displeasing if the least common parent is situated at a high level or even is the overall root service of the service hierarchy. However this could be the consequence of a badly structured service hierarchy. As we will see later (see Section 4.8, *Combination of services on basis of the service relationships*) this way of handling persistent data also goes along well with the semantics of our notational techniques. As mentioned above, conflicts might occur between services if they want to write the same variable at the same time. We will also come back to this point in Section 4.8 (*Combination of services on basis of the service relationships*) by explicitly introducing conflict solving for this situation in our methodology.

Please note that the most appropriate way of handling persistent data would be to make it visible to only those services which actually need the data. However this is not possible with our notational techniques.

Remarks

The service hierarchy is only an intermediate result. It serves as a basis for the service graph. Therefore, the final result of this methodological step is the service graph.

Note that by organizing services hierarchically, further atomic services can be identified. The high-level structure of a service graph can serve as a service identification framework the usage of which is suggested in [Kang et al., 2002a].

The set of complex horizontal service relationships is infinitely large. However, it might be stable for a specific domain (e. g. automotive systems). Most probably, after the development of several products within the same domain, a catalog of complex horizontal service relationships can be created. Thus, the horizontal service relationships can be reused. (See also comments on product line development in Section 7.3, *Outlook*.)

4.6.2. Notational technique(s)

As already mentioned in the previous subsection, we make use of *directed, hierarchical graphs* in order to represent our vertical and horizontal service relationships. Figures 4.9 and 4.10 show directed graphs with the first one containing only the vertical relationships (so-called *service hierarchy*) while the second one contains both the vertical and horizontal service relationships (so-called *service graph*).

Each node represents a service behavior. The leaves of the graph stand for the behavior of the atomic services. The inner nodes represent the behavior of the combined sub services. Edges visualize service relationships. In the service hierarchy, directed edges point to sub services. In the service graph we also have horizontal arrows representing the horizontal service relationships.

The nodes of the hierarchy contain the names of the services, respectively. Nodes can also contain the name of persistent data (data: <name>) indicated by the keyword "data". In that case the data is visible to all sub services. The edges which are not labeled represent the sub service relationship (vertical relationships). All other relationships (horizontal relationships) are labeled with the name of the relationship (and its parameters if appropriate).

Each complex horizontal service relationship has a textual description. Furthermore

- if need be - it has to be specified which parameters are required by the complex horizontal service relationship and what the codomain of the parameters is, respectively.

The textual descriptions of the complex horizontal service relationships can be listed in tables (see Table 4.8). Each complex horizontal service relationship has a name and an abbreviation. Furthermore, the arity has to be specified indicating between how many services the relationship exists. Moreover, it has to be specified whether the relationship is symmetric or asymmetric. Textual descriptions of the relationship and its parameters also have to be given.

Table 4.8.: Description of complex horizontal service relationships

Name	Abbr.	Arity	(A)Symm	Description	Parameters
...	...	...	...	...	...

It is not necessary to list the combined services in a table, too, like we suggested for the atomic services. The combined services serve as a container for its sub services and the information which sub services belong to which super service is already captured in the service hierarchy (and thus also in the service graph).

4.6.3. Methodological steps

The aim is to create one service hierarchy for the system under specification. For medium to large scale systems this can be quite difficult. We suggest a meet-in-the-middle approach. A meet-in-the-middle approach is a mixture of a top down approach and a bottom up approach. For our service hierarchy we have the following starting points concerning the service hierarchy: The root is the overall system functionality; the leaves are the atomic services (see Section 4.4, *Identification of atomic services*). The inner nodes represent the combined services. It may be helpful to first create sub trees of the hierarchy and combine them afterward. Such a sub tree would contain the services of a sub system for example.

Figure 4.11 contains an overview of the activities performed during the identification of service relationships.

First the service hierarchy has to be created. To that end, the leaves of the service hierarchy are the atomic services and the root represents the overall system functionality. These activities can be performed in parallel.

The determination of the vertical and horizontal service relationships is tightly coupled as the horizontal service relationships motivate the introduction of combined services (super services). Therefore, the step of the identification of the service relationships is an iterative procedure.

The combined services (super services of several sub services) which are explicitly mentioned in the requirements are identified. Vertical sub service relationships are introduced between the combined services and their sub services. Thus, the service hierarchy is created. Additionally, the horizontal service relationships (both basic and complex ones) are determined. The service hierarchy is enriched by the horizontal service relationships. The result is the service graph. Each complex horizontal service relationship has to be described in table form.

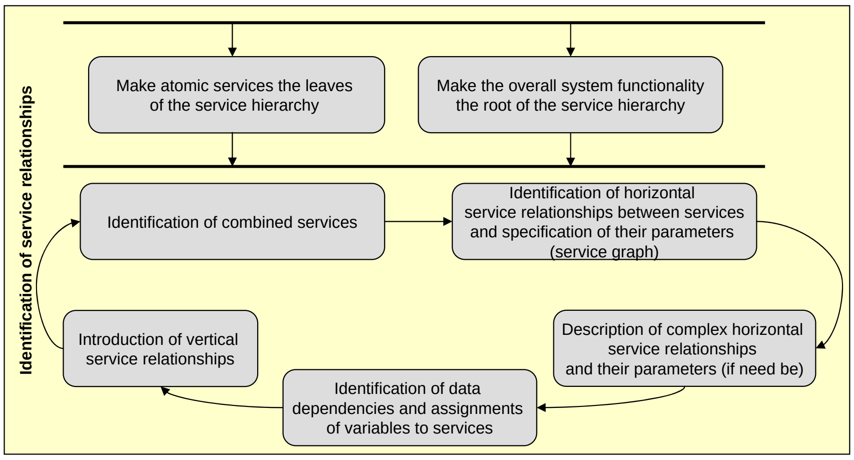
Identification of service relationships
Make atomic services the leaves of the service hierarchy
Make the overall system functionality the root of the service hierarchy
Identification of combined services
Identification of horizontal service relationships between services and specification of their parameters (service graph)
Introduction of vertical service relationships
Description of complex horizontal service relationships and their parameters (if need be)
Identification of data dependencies and assignments of variables to services

Figure 4.11.: Identification of service relationships (activity diagram)

Additionally, data dependencies between services operating on the same data have to be identified. This goes along with the assignment of data to the least common parent.

4.6.4. Application to the case study

In this subsection, we show how we determine the service relationships of our running example step by step. Figure 4.12 shows a first version of the service hierarchy. The root (shaded oval) represents the overall system functionality. The leaves (filled ovals) represent the atomic services (see Section 4.4, *Identification of atomic services*). Due to reasons of clarity, we omit the assignment of data in this and the following figures. The services MEMORY FUNCTIONALITY VIA BUTTON and MEMORY FUNCTIONALITY VIA CAR KEY would be assigned with the persistent data POSBACK, POSDIST, POSFRONT, and POSREAR. Each of the eight adjustment services would also have a data entry specifying the data for the respective position.

First, we introduce the combined services which are explicitly contained in the requirements: Requirement 3 determines the introduction of the services MANUAL ADJUSTMENT and ADJUSTMENT BY MEMORY. The Requirements 1a,1b,1c, and 1d lead to the introduction of the services ADJUSTMENT OF BACK, ADJUSTMENT OF DISTANCE, ADJUSTMENT OF REAR AREA, and ADJUSTMENT OF FRONT AREA.

Respective vertical service relationships are introduced. Some of the atomic services are not related hierarchically so far (see atomic services at the bottom of Figure 4.12). This will be done next, when the horizontal service relationships are identified.

The horizontal service relationships lead to the introduction of combined services and thus more vertical service relationships. They are explained in the following. First, we only introduce the horizontal service relationships (and no further vertical service relationships). Figure 4.13 shows the service graph of our running example. Due to reasons of clarity, the parameters of the complex horizontal relationships have been omitted in the figure.

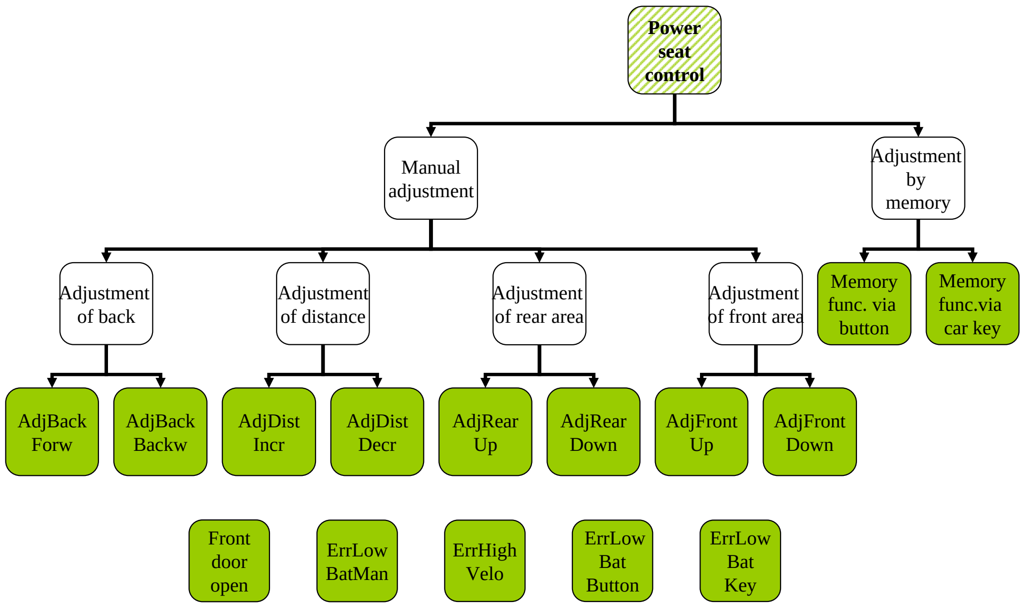

Figure 4.12.: Service hierarchy of the power seat control system (service hierarchy)

Requirement 5 leads to the ENABLE and DISABLE relationships between the services FRONT DOOR OPEN and MANUAL ADJUSTMENT. Requirement 9 demands for the MAX2PAR relationship between the adjustment services. (The MAX2PAR relationship is described below.) The introduction of the XOR relationship between the leaves of the MANUAL ADJUSTMENT service is a result of Requirement 8. (The XOR relationship is also described below.) The ENABLE and DISABLE relationships between the services MANUAL ADJUSTMENT and ERRLOWBATMAN are a consequence of Requirement 10.

Requirements 11, 14, and 16 lead to the assignment of the remaining four ENABLE and DISABLE relationships.

The memory services both operate on the same persistent data, namely the POSBACK, POSDIST, POSFRONT, and POSREAR. The data dependency between these two service visualizes this fact.

The service hierarchy can be obtained from the service graph by omitting the horizontal service relationships.

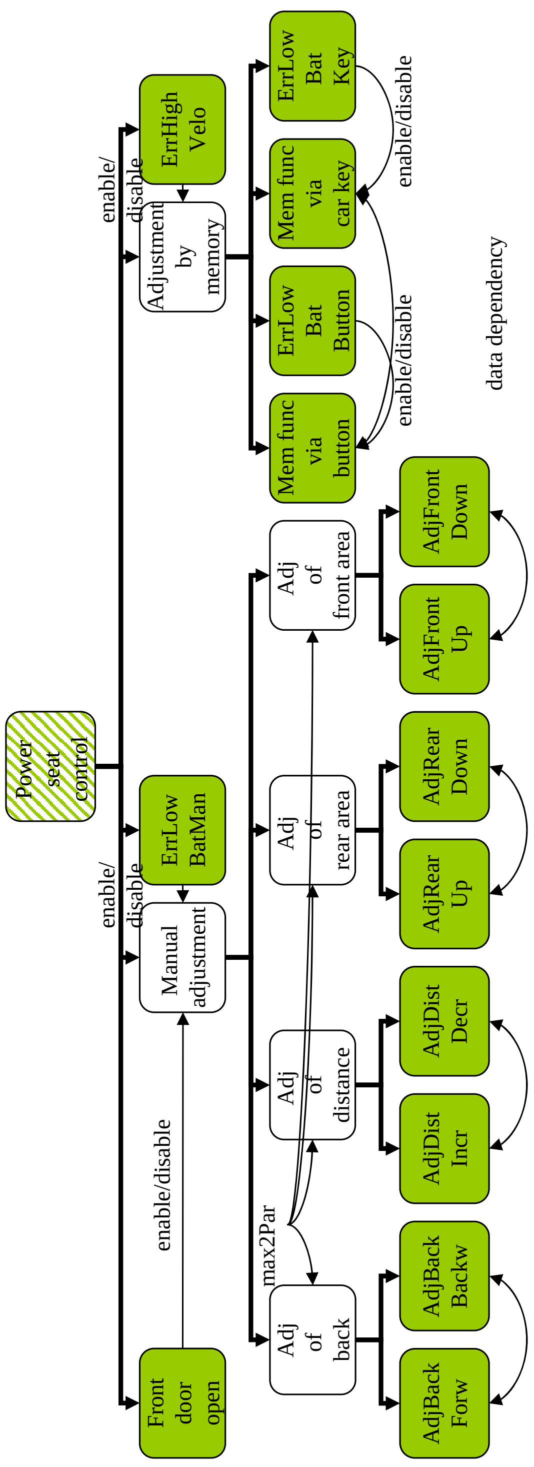

Figure 4.13.: Service graph of the power seat control system (service graph)

Table 4.9.: Description of the complex horizontal service relationships of the power seat control system

Name	Abbr.	Arity	Asymm	Description	Parameters
mutual exclusion	XOR	n	symm	If services S1, S2, ..., Sn mutually exclude each other, only one service can be active at once.	PAR ∈ {<NONE>,S1,S2,..., SN} specifies which service to execute in case two or more services are called at the same time. ANOTHER ∈ {<RUNNING>,<FIRST>,<STARTED>,PRIORITY(SX,SY)} defines what to do in case a service is already being executed and another one is called. <RUNNING> means that the running service is continued to be executed. <FIRST> defines that the service which is already running is executed further. <STARTED> means that the newly called service is executed and the running service is stopped. PRIORITY(SX,SY) is a function introducing an ordering on the services which service has a higher priority.
maximal two parallel	MAX2PAR	n	symm	If there is a MAX2PAR relationship between n services, only 2 of the services can be active at once.	PAR ∈ {<NONE>,PRIORITY(SX,SY)} defines which service is to be executed if more than two services are called at the same time. ANOTHER ∈ {<NONE>, PRIORITY(SX,SY)} specifies what to do if two services are already running and another service is called.

As far as our running example is concerned we do not know to which values we should set the parameters. This is not covered by the requirements. This is a good example to show how our model-based approach helps at identifying missing requirements (although this is not in the main focus of the thesis at hand).

We just decide to set the parameters as follows.[12] The XOR relationships between the adjustment services get the parameter ANOTHER = <FIRST>. The parameter PAR is set to the FORWARDS, INCREASING, and UP services, respectively. The service relationship MAX2PAR has the parameter PAR set to <NONE> and the parameter ANOTHER set to <NONE>, too.

4.7. Formal specification of atomic services

So far, we took a quite informal look onto the system. During the remaining steps of the methodology, the system functionality is specified formally. The first step is to specify the modular, atomic services formally. In this section, we show how this is done. Based on the service relationships identified in the previous step, the formal specifications of the modular services will be combined in the following section (see Section 4.8, *Combination of services on basis of the service relationships*).

4.7.1. Concepts

This step of the methodology bridges the gap between informal to formal service descriptions. The informal (textual) service specifications are transformed into formal service models, respectively.

In this step, only the atomic services (i. e. the leaves of the service graph) are formalized. The formalization of the super services, particularly of the root service which is the overall system behavior will later be obtained as the result of the combination of the atomic services. When formalizing the atomic services, the effects resulting from the service relationships is not taken into consideration. Only the modular behavior, as if the service was alone in the system and no other services had an effect on the service is specified formally. We call this behavior the *core behavior* of a service.

Some atomic services may be quite trivial (e. g. the service FRONT DOOR OPEN). However, as we will see later - when combining the services in Section 4.8 (*Combination of services on basis of the service relationships*) - the behavior will become more comprehensive. Therefore we formalize all atomic services, although it may not make sense for the trivial services with the knowledge we have at this stage of the methodology.

Another possibility would be to distinguish between trivial services ("events" or "preconditions/guards" like the FRONT DOOR OPEN service) and non-trivial services (like the service MEMORY FUNCTIONALITY VIA CAR KEY). However, as we will see later, for our methodology this distinction is not necessary as trivial services ("events" or "preconditions/guards") and non-trivial services are treated in the same way. Concerning habitual language use (the FRONT DOOR OPEN service usually would be

[12]Of course, during the development of a real system, these requirements would have to be elicited later.

called a feature/service, too) and in order to keep the number of concepts of our methodology as small as possible we choose to not make this distinction.

For each service both the syntactic and semantic interface have to be formally specified. The syntactic interface (logical channels and logical actions) is comprised of those logical input and output actions (see Section 4.5, *Logical syntactic system interface*) which the service makes use of. Thus, the logical interface of a service is a *sub type* of the logical interface of the overall system. Here, we make use of the concept of interface sub typing as formally founded in [Broy, 2007]. Informally spoken, a syntactic interface I1 is a subtype of the interface I2, if the channels of I1 are a subset of the channels of I2 and if the messages ("actions" in our context) possibly sent on the channels of I1 are a subset of the messages possibly sent on the channels of I2. I1 therefore is a projection of I2. This refers to both the input and output channels of the syntactic interface.

As far as the semantic interface is concerned, each service is specified by relating logical input actions to logical output actions. In our methodology this is done by I/O automata (see Section 4.7.2 (*Notational technique(s)*)). Automata are an appropriate means for specifying reactive behavior. Automata (amongst others) have an operational semantics [Olderog, 1986]. Note that it might not make sense to specify each service by an operational specification technique. For example a service which gets a list and returns a sorted list may be better specified in an assumption/guarantee style. The question which service should be specified by which technique is not scope of this thesis and should be investigated in future work.

Formally, a service can be seen as a *finite automaton*. It relates a set of input actions to a set of output actions. A service can enter different control states (which structure the service execution).[13] Transitions between control states can be considered to be service steps. Furthermore, a service has exactly one initial state. When the service is in the initial state and the triggering input action occurs, the service starts the service execution. A service does not need to have one or more end states as the ending of a service execution might result in the initial state again. Thus, end states are optional to a service specification. A service may operate on local data. In order to be able to speak about preemption (and also priorities) we also need to capture the time aspect of a service execution. We assume that the service specifications are based on an equidistant time grid. Transitions between control states of services (i. e. service steps) take one time unit. Thus, if we introduce an additional control state, we also introduce an additional delay. We will illustrate this fact in Section 4.7.2 (*Notational technique(s)*) by a small example and with the help of our notational techniques.

Furthermore, a service step (transition) can not be interrupted. A service execution can only be interrupted at control states. Thus, if we want to express that a service can be interrupted at some point in the service execution, we have to insert an additional control state. Analogously, if we want to express that the service execution can not be interrupted at some point, no control state should be introduced.[14]

[13]As the service relationships, and thus the service combination, may depend on the execution status of services (see Section 4.8, *Combination of services on basis of the service relationships*) the state view of automata is very helpful.

[14]Note that we will later loosen this demand by the introduction of "secure interrupts" in Section 5 (*Extension of basic service relationships*).

As mentioned above, services might operate on persistent data. During the identification of atomic services, persistent data is specified informally (see Section 4.4, *Identification of atomic services*). When formalizing services we have to define a *data type* for each persistent data. Persistent data can either be of a basic data type (like Integer, Real, or String) or of a complex data type. In latter case this complex data type first has to be defined for example by means of AutoFOCUS Data Type Definitions (DTDs) [Huber et al., 1997]. Then, the persistent data can be specified formally. Each data is given a data type, a unit (if appropriate), a range, and an initial value. Note that this formal data specification holds for all services which make use of the persistent data.

Having a closer look at the services of the running example, we observe that some services have the same "behavioral pattern". For example all manual adjustment services (ADJUSTMENT OF THE BACK FORWARDS, ADJUSTMENT OF THE BACK BACKWARDS, ADJUSTMENT OF THE DISTANCE INCREASING, ...) exhibit the same kind of behavior: On receiving the driver's wish to adjust the seat, the respective motor is controlled. Of course, a formal specification could be given for each service. However, this effort can be reduced by introducing a *service pattern* which can be instantiated with different actions. An example for such a service pattern is given in Section 4.7.4 (*Application to the case study*).

4.7.2. Notational technique(s)

In order to formally specify the syntactic and semantic interface of atomic services, we use System Structure Diagrams (SSDs) and State Transition Diagram STDs, respectively (see Sections 3.1, *System Structure Diagrams (SSDs)* and 3.2, *State Transition Diagrams (STDs)*).

We combine these two diagrams and display it in one graphic (see Figure 4.14). Sometimes, the behavior of an atomic service is comprised of parallel sub functionality. For example, the service MEMORY FUNCTIONALITY VIA BUTTON adjusts the four dimensions of the seat position in parallel. In that case, it is better to decompose an SSD into further SSDs with contain the parallel functionality. The parallel functionality is then specified by an STD, respectively. Figure 4.19 shows an example.

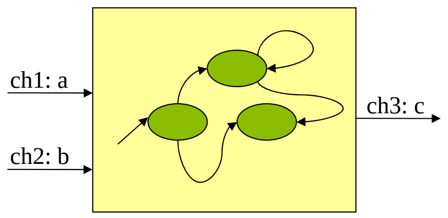

Figure 4.14.: Formal specification of atomic services (SSD + STD)

A service - in our context - is a partial behavior. Consequently, it is (in general) modeled by a partial STD. Partiality in this context means, that it is not (necessarily) specified how the system shall react in each situation (state) if a certain action is received. In that case, we make no statement about the behavior of the service.

As mentioned in the previous paragraph, transitions (per definition) take one time

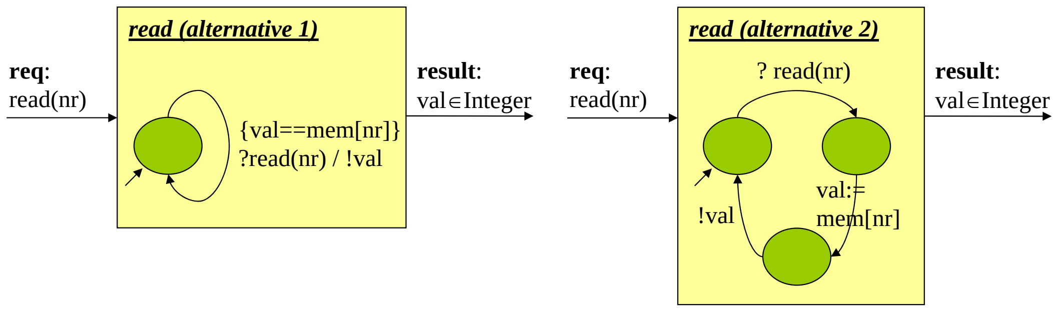

Figure 4.15.: Two alternative specifications of a simple service (SSD + STD)

tick. Thus, if we introduce an additional control state, we also introduce an additional delay. Figure 4.15 shows two alternative specifications of a simple read service. On receiving a read request, it calculates the value stored, and issues it. In the left specification these steps are all done in one transition. Consequently, the read request can not be interrupted and takes one time tick. The specification in the right hand side is different. For each step, one transition is introduced. Thus, the service can be interrupted after each service step (at each control state) and takes three time ticks. Additionally, a service request can only be handled every third tick (when the service is in the initial state). This problem can be eliminated by scaling the time ticks of services accordingly. For example, for the read service of the right specification we could define that a tick is only a third of a global time tick. One has to be aware of this implicit information when specifying a service with an STD. For reasons of simplicity, we do not take into consideration timing problems in the remainder of this thesis. We leave topics like the scaling of time ticks within service specifications for future work (see Section 7.3, *Outlook*). For our work it is (just) important to be able to speak of an interruption of a service. To that end, we introduce control states to the service specification.

Note that due to the semantics of STDs (see Section 3.2.3, *Semantics of STDs*), the system reaction is left open for underspecified transitions. This goes along with our understanding of services which make no statement about how the system shall react in underspecified cases.

In Section 3.2.2 (*Priority concept for competing transitions*) we introduced the priority concept for competing transitions. For the formal specification of atomic services, only the priorities 0 to 4 are allowed to be used. We will give a motivation for that in Section 4.8 (*Combination of services on basis of the service relationships*).[15]

Persistent data is formally specified by enlarging the data table which was already suggested for the informal specification (see Table 4.1). The result looks as shown in Table 4.10.

4.7.3. Methodological steps

The activities of this step of the methodology are shown in Figure 4.16. First the syntactic interface is formally specified for each atomic service. To that end, an STD

[15]Priority 5 will be used to handle feature interaction.

Table 4.10.: Formal specification of persistent data

Abbreviation	Name	Description	Data type	Unit	Range	Initial Value
...	...	...	...	...	...	...

is created for each service. If the service operates on persistent data (and the data has not formally been specified so far), the data is formally specified. This includes the definition of complex data types if need be. The assignment of the data to services has already been done during the identification of service relationships. The semantic interface of the core behavior[16] of each service is specified by an STD.

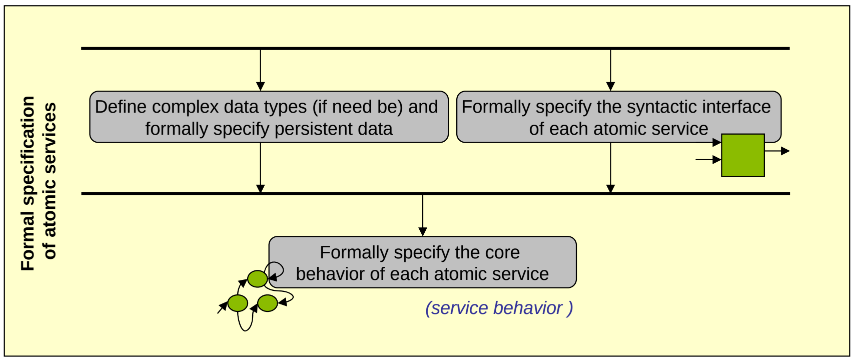
Formal specification of atomic services
Define complex data types (if need be) and formally specify persistent data
Formally specify the syntactic interface of each atomic service
Formally specify the core behavior of each atomic service
(service behavior)

Figure 4.16.: Formal specification of atomic services (activity diagram)

4.7.4. Application to the case study

For our running example, we identified the following atomic services (see Section 4.4.4, *Application to the case study*):

- ADJUSTMENT OF THE BACK FORWARDS
- ADJUSTMENT OF THE BACK BACKWARDS
- ADJUSTMENT OF THE DISTANCE INCREASING
- ADJUSTMENT OF THE DISTANCE DECREASING
- ADJUSTMENT OF THE REAR AREA UP
- ADJUSTMENT OF THE REAR AREA DOWN
- ADJUSTMENT OF THE FRONT AREA UP
- ADJUSTMENT OF THE FRONT AREA DOWN
- FRONT DOOR OPEN
- MEMORY FUNCTIONALITY VIA BUTTON

[16]As mentioned above, the core behavior of a service is the modular "characteristic" behavior of a service which is not influenced by other services.

- MEMORY FUNCTIONALITY VIA CAR KEY
- ERR HIGH VELOCITY
- ERR LOW BATTERY BUTTON
- ERR LOW BATTERY CAR KEY
- ERR LOW BATTERY MANUAL

In the following, we formally specify their syntactic and semantic interface by an SSD and an STD, respectively. As we will see, the initial states of the services are not always defined by the requirements. This points at missing requirements which would have to be elicited. As far as our running example is concerned, we will choose the initial states as we think is reasonable.

Due to reasons of clarity we first formally specify the persistent data of our running example. The formal service specifications of the atomic services can then make use of the formal data specifications. Table 4.11 shows the result.

Table 4.11.: Formal specification of the persistent data of the power seat control system

Abbr.	Name	Description	Data type	Unit	Range	Init.
posBack	...	The variable contains the position of the back as saved by the user management (10 represents the middle position; 1 is the position with the back most forward; 21 is the position with the back most backward).	Integer	-	[1..21]	1
posDist	...	...	Integer	-	[1..21]	1
posFront	...	...	Integer	-	[1..21]	1
posRear	...	...	Integer	-	[1..21]	1

The manual adjustment services

The manual adjustment services (the first eight services of the listing above) all operate according to the same pattern: If the driver expresses his/her wish to move the seat in a direction, the motor is controlled. Figure 4.17 shows this pattern. It has to be instantiated for each manual adjustment service. For example for the service ADJUSTMENT OF THE BACK FORWARDS this pattern has to be instantiated with <chI>=MOVE_BACK_BACKW, <direction>=BACK_BACKW, <chO>=MOTOR_BACK_BACKW, <command>=M_BACK_BACKW, and <state>=INIT. (As the other services are instantiated analogously, we omit their formal specification.)

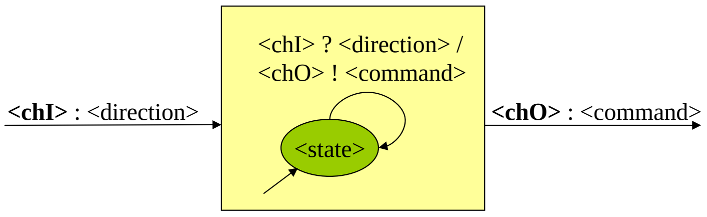

Figure 4.17.: Formal specification of the MANUAL ADJUSTMENT services by a pattern (SSD + STD)

The FRONT DOOR OPEN service

As mentioned above, some atomic services are quite trivial. The FRONT DOOR OPEN service falls into this category. As the core behavior of the service does not have an output behavior, we just specify its syntactic interface at this point (see Figure 4.18).[17]

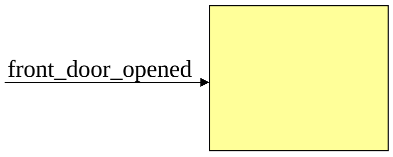

Figure 4.18.: Formal specification of the FRONT DOOR OPEN service (SSD)

The MEMORY FUNCTIONALITY VIA BUTTON and MEMORY FUNCTIONALITY VIA CAR KEY services

The MEMORY FUNCTIONALITY VIA BUTTON service is more complicated. As demanded in Requirement 13, the seat has to be moved into the previously saved position as fast as possible. Therefore, all movements have to be carried out simultaneously.

The idea behind the specification of the service behavior is the following: On receiving the command ADJUST_BY_BUTTON, the seat is moved until the actions sent by the position sensor match the values saved. (Time delays are not taken into consideration.)

As specified in 4.5.4 (*Application to the case study*), the logical input actions POS_BACK, POS_DIST, POS_REAR, and POS_FRONT represent the current position of the back, the distance, the rear area, and the front area of the seat by Integer values, respectively.

The variables POSBACK, POSDIST, POSREAR, and POSFRONT (of type Integer) contain the previously saved positions and were already introduced during the identification of the service relationships. Note that a service SAVE CURRENT SEAT POSITION would write these variables. However this service is not specified in this thesis.

[17]When combining the services later (see Section 4.8, *Combination of services on basis of the service relationships*) the core behavior of the FRONT DOOR OPEN service will be adapted and thus become non-trivial.

Figure 4.19 shows the syntactic (SSD) and semantic (STD) interface of the MEMORY FUNCTIONALITY VIA BUTTON service. On receiving the input action ADJUST_BY_BUTTON, the four directions are controlled in parallel. For each direction, an SSD and STD is introduced, respectively. The seat is in the right position when all current seat positions (sent by the position sensor by the actions POS_BACK, POS_DIST, POS_REAR, and POS_FRONT) match the saved values (represented by the variables POSBACK, POSDIST, POSREAR,.

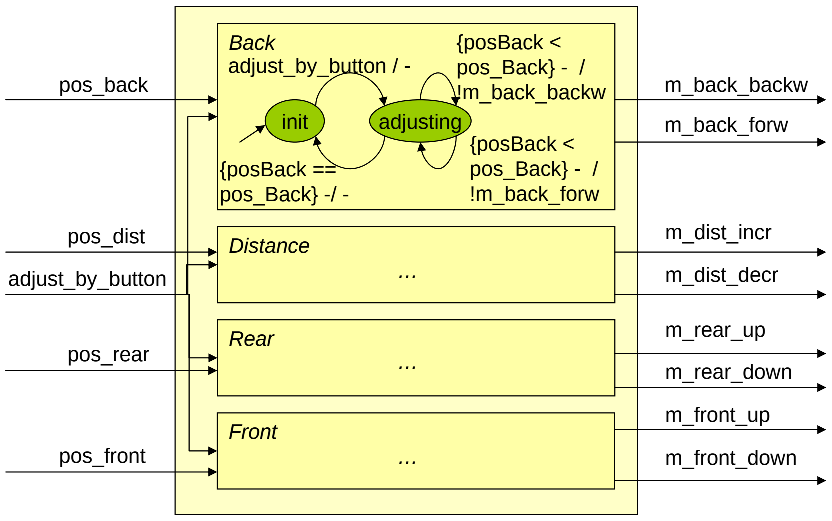

Figure 4.19.: Formal specification of the MEMORY FUNCTIONALITY VIA BUTTON service (SSD + STD)

The MEMORY FUNCTIONALITY VIA CAR KEY services can be designed similarly.

The ERROR LOW BATTERY services

Figure 4.20 shows the formal specification of the service ERR LOW BATTERY MANUAL. The other error services (ERR LOW BATTERY BUTTON and ERR LOW BATTERY CAR KEY) can be specified analogously. (Their specification is omitted here.) gain, we could make use of a common service pattern as all three error services expose the same behavior.

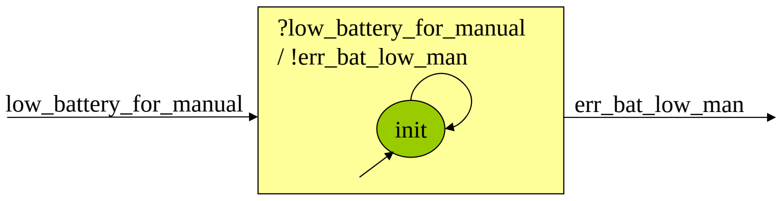

Figure 4.20.: Formal specification of the ERR LOW BATTERY MANUAL service (SSD + STD)

The ERR HIGH VELOCITY service

The ERR HIGH VELOCITY service is a trivial service. Its syntactic interface is shown in Figure 4.21.

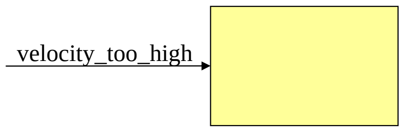

Figure 4.21.: Formal specification of the ERR HIGH VELOCITY service (SSD)

4.8. Combination of services on basis of the service relationships

So far we have (formally) specified the services in a modular fashion. In order to obtain the overall system functionality, we have to combine the services, i. e. we also have to model their interplay.

4.8.1. Concepts

In Section 4.6 (*Identification of service relationships*) we distinguished between vertical and horizontal service relationships. As far as the combination of services is concerned, the vertical relationships (which make up the service hierarchy) define the order in which the services are combined (bottom up strategy) and the horizontal relationships define the way of combining them. To be more precise: the horizontal service relationships define how the core behavior of the services has to be modified in order to handle the influences of other services.

For the combination of the modular service specifications we use a bottom up strategy: The combination of atomic services leads to more comprehensive services, so-called *combined services*. These combined services are again combined until the overall system functionality is obtained. Concerning the service graph (see Figure 4.10), the leaves are combined to get the behavior of inner nodes. Inner nodes are combined again until the overall system functionality which is represented by the root is obtained.

For each combined service (note that our notion of service is relative, see Section 4.1.1, *Underlying system model*), we have to specify the syntactic and semantic interface, too.

The syntactic interface can be obtained quite easy. It is the combination (least super type, see [Broy, 2007]) of the syntactic interfaces of its sub services. The syntactic interface of the root thus has the logical syntactic interface of the overall system. Given the modular service specifications, the logical syntactic interface of their combined behavior (their super service) can be generated automatically.

To obtain the semantic interface of the combined behavior we first have to make some considerations. Services are combined based on the service relationships between them. To that end, we have to investigate what effects a horizontal service relationship has on the behavior of a service. Note that the behavior of all services "involved" in a horizontal service relationship is affected. For example if a service S1 influences a service S2 (see Figure 4.22),

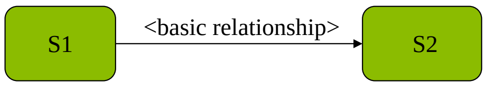

Figure 4.22.: Influencing and influenced service (schematic picture)

- the modularly specified behavior (core behavior) of S1 (influencing service) has to be modified to indicate at what point the service S2 shall be influenced and
- the modularly specified behavior (core behavior) of S2 (influenced service) has to be modified to handle the influence of S1.

This holds analogously for service relationships of arbitrary arity.

In the following we first show how each *basic service relationship* affects the *core behavior* (modular service specification) of a service. Then we will show how complex horizontal service relationships influence service specifications.

Realization of basic service relationships

In Section 4.6.1 (*Concepts*), we introduced a set of basic relationships (ENABLE, DISABLE, INTERRUPT, CONTINUE, RESET, and INDEPENDENT) that influence a service behavior.[18]

In the following we show how the effect of the basic service relationships can be realized by the influenced and the influencing service. As we model the syntactic and semantic interface of services by means of SSD and STD, respectively, we show how the SSD and the STD have to be modified in order to realize the influence of the basic service relationships. For each basic service relationship we give a *standard control interface*. A service which is affected by a basic service relationship (e. g. DISABLE) has to implement the respective *standard control interface* in order to handle this influence. If a service is affected by several basic service relationships, it has to implement all respective standard control interfaces.

Put in other words, we give schemata how to transform the *core behavior* of a service into the so-called *modified behavior* of a service. The latter is the service behavior as exposed in the overall system functionality.

Realization of the basic service relationship DISABLE Figure 4.23 illustrates the realization of the basic service relationship DISABLE. The upper part of the graphic contains the service relationship as used in the service graph. The lower part contains the service specification after the realization of the service relationship. The service S2 implements the standard control interface of the DISABLE service relationship.

[18]The INDEPENDENT relationship does not influence other services.

Let INPUT-CH1, ... INPUT-CHN be the logical input channels and OUTPUT-CH1, ... OUTPUT-CHM be the logical output channels of the service. The syntactic interface (SSD) is enlarged by a channel (named S1_S2, indicated by a dashed arrow) and by the logical input action DISABLE. This action is sent by the influencing service S1 when the influenced service S2 has to be disabled.

As far as the semantic interface of the influenced service is concerned (represented by the STD), a hierarchical automaton is introduced. On the highest level, the automaton has the two states DISABLED and ENABLED. The semantics are the following: If the service is in the state ENABLED, the service can be executed (according to the modular specification of its core behavior as specified in Section 5.4.3, *Formal specification of modular services*). If the service is in the state DISABLED, it can not be executed. No output is produced even if the input actions triggering the service are received. The transition ?X / !NIL of Figure 4.23 is a universal transition which accepts each input on each input channel (except those inputs for which a transition is explicitly defined, for example ?ENABLE / ..., see later). NIL is the empty value which represents the absence of a message on each output channel of the universal transition. Thus the system reaction is not left open, but it is specified that the service produces no output. On receiving the input action DISABLE, the service leaves the state ENABLED and enters the state DISABLED. As the transition leading from the state ENABLED to the state DISABLED has PRIO=5 (and transitions of the modular service specification can only have priorities ranging from 0 to 5, see Section 4.7.2, *Notational technique(s)*), this transition is fired. In case local data has been changed, it must additionally be set back to the initial value. (This might lead to conflicts, see later.) This can be done by specifying the postcondition (e. g. ?DISABLE/!NIL {VAL:=INITVAL}) accordingly.

As the channel S1_S2 aims at a special purpose, namely to connect influencing and influenced service(s), it is formated differently. We make use of dashed arrows instead of solid arrows to represent these kinds of special channels. However, as mentioned in Section 3.1 (*System Structure Diagrams (SSDs)*) they have the same semantics as the solid arrows.

Note, we decided to enable the service per default. (The ENABLED state is the initial state of the automaton.) If demanded differently by the requirements this can of course be changed.

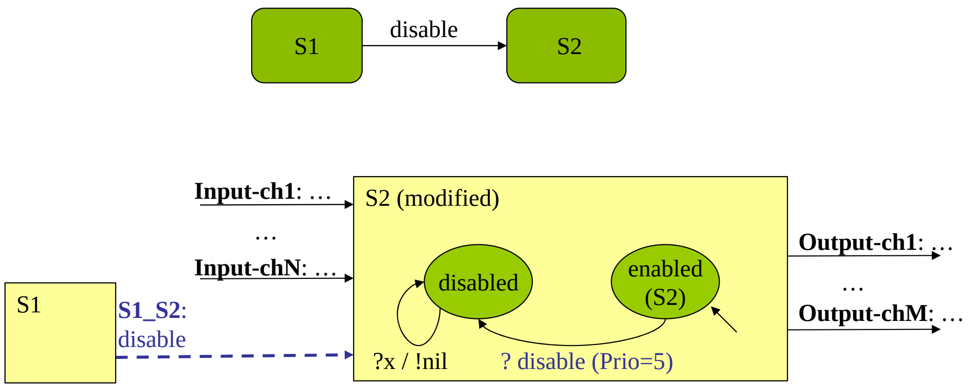

Figure 4.23.: Standard control interface of the basic service relationship DISABLE (SSD + STD)

Realization of the basic service relationship ENABLE The standard control interface of the basic service relationship ENABLE is shown in Figure 4.24. It basically can be seen as the inverse behavior of the DISABLE service relationship. If a service is currently disabled, it transits to the state ENABLED on receiving the action ENABLE.

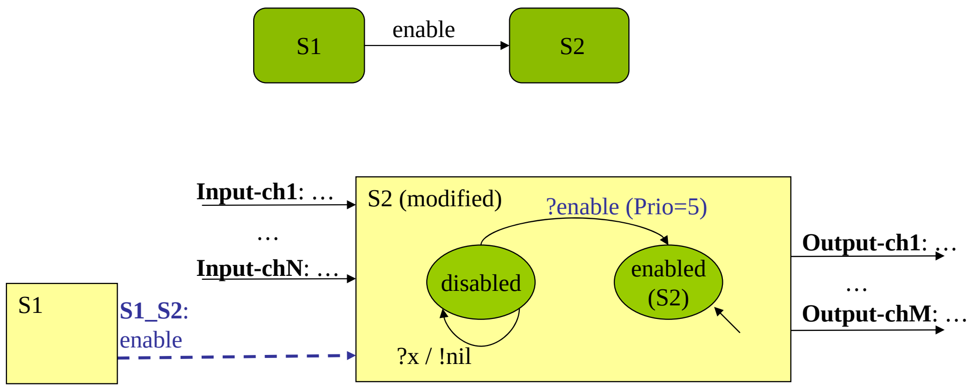

Figure 4.24.: Standard control interface of the basic service relationship ENABLE (SSD + STD)

Realization of the basic service relationship INTERRUPT The realization of the basic service relationship INTERRUPT is shown in Figure 4.25. Again, the logical syntactic interface is enlarged by a channel (S1_S2) and by the logical input action INTERRUPT.

The automaton is a high level automaton having the states RUNNING[19] and INTERRUPTED. In the state RUNNING the service can be executed as specified modularly. On receiving the action INTERRUPT the service exits the state RUNNING and goes into the state INTERRUPTED. In this state, the empty value NIL is sent on all output channels.

As noted before, the basic service relationships DISABLE and INTERRUPT are semantically equivalent. However, for the basic service relationship CONTINUE (see proceeding paragraph) it is necessary to "remember" the state in which the service specification was left. For the ENABLE relationship this is not necessary. Therefore, we explicitly distinguish between DISABLE and INTERRUPT.

Realization of the basic service relationship CONTINUE Figure 4.26 contains the standard control interface of the basic service relationship CONTINUE. Being caused to continue its execution (by receiving the action CONTINUE) the service enters the history state of RUNNING again (indicated by the encircled H). Due to the history state of the service, the service execution can be continued at the point where it was interrupted.

Realization of the RESET relationship The realization of the RESET relationship is shown in Figure 4.27. This time we introduce one high level state (containing

[19]Please note, that the service can also be in the state ENABLED (see above) when being "interrupted".

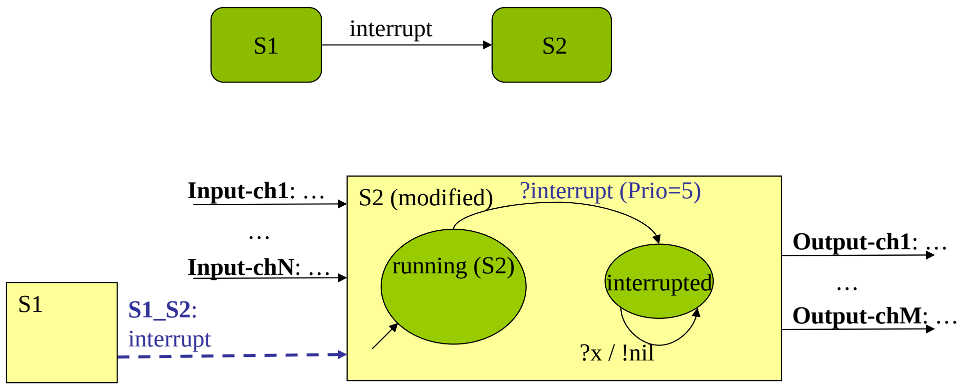

Figure 4.25.: Standard control interface of the basic service relationship INTERRUPT (SSD + STD)

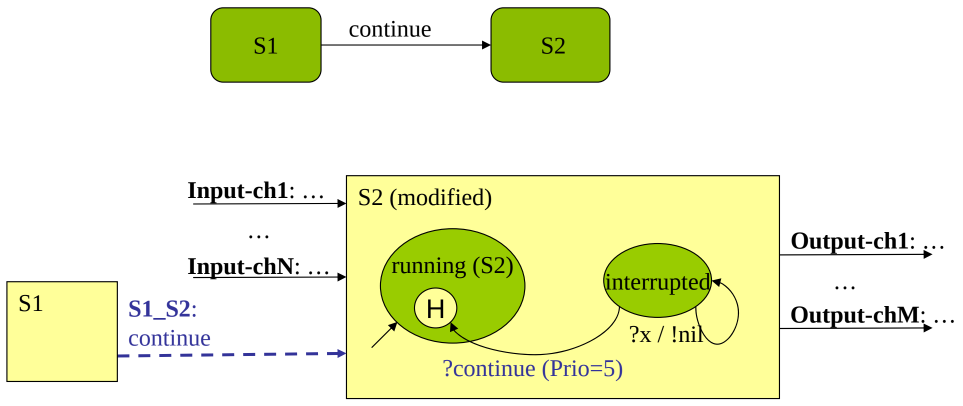

Figure 4.26.: Standard control interface of the basic service relationship CONTINUE (SSD + STD)

the core behavior of service S2) and a transition starting and ending at this high level state. The syntactic interface is enlarged by the logical input action RESET. On receiving this action, the service performs the transition which causes the service execution to stop and the service to enter its initial state. In case the service makes use of data, these data must additionally be reset to the initial value.[20] This can be achieved by specifying the postcondition of the transition accordingly (?RESET/!NIL {VAL:=INITVAL}).

Realization of the INDEPENDENT relationship If two services are independent, the modular specifications do not have to be modified. Their specifications is simply combined in parallel.

As the actions ENABLE, DISABLE, INTERRUPT, CONTINUE, and RESET play a special

[20]Analogously to the DISABLE service relationship.

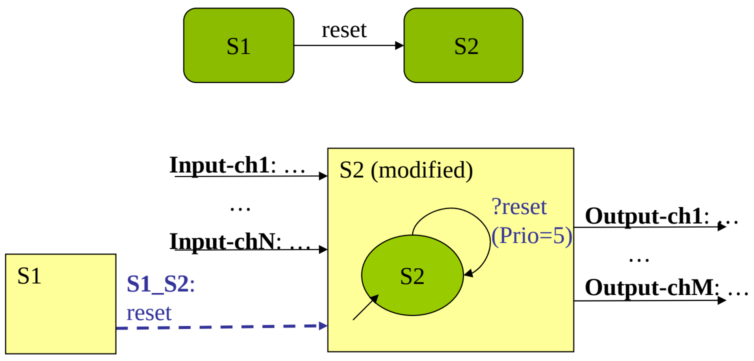

Figure 4.27.: Standard control interface of the basic service relationship RESET (SSD + STD)

role in our methodology (namely to realize the basic service relationships), we give them a special name: *basic actions*.

In the preceeding paragraphs, we showed how the specification of the influenced services have to be modified in order to handle the basic service relationships. As mentioned above the modular specifications of the influencing services have to be modified, too. To that end, the influencing services have to send the respective actions (ENABLE, DISABLE, INTERRUPT, CONTINUE, and RESET) at the point of the service execution where appropriate. This time however, the syntactic *output* interface is enlarged. (See Section 4.8.4, *Application to the case study* for examples.)

Figure 4.28 schematically shows how the specification of the influencing service is modified.

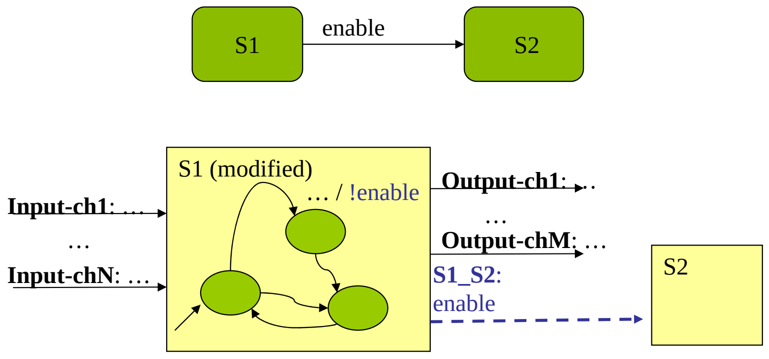

Figure 4.28.: Realization of the ENABLE relationship by the influencing service (SSD + STD)

Figure 4.29 shows a modular service specification implementing the basic service relationships ENABLE, DISABLE, INTERRUPT, CONTINUE, and RESET. Note, that the result is a partial behavior again. It is not specified how the service has to behave if a DISABLED action occurs in the INTERRUPTED state (and vice versa). This can not be said in general but has to be specified specifically for each service. Therefore it is not demanded by the standard schema.

For the communication between services (in order to realize the service relationships) we introduce channels between the services. As convention we name the channels S1_S2 if the directed channel starts at S1 and leads to S2. As all channels should

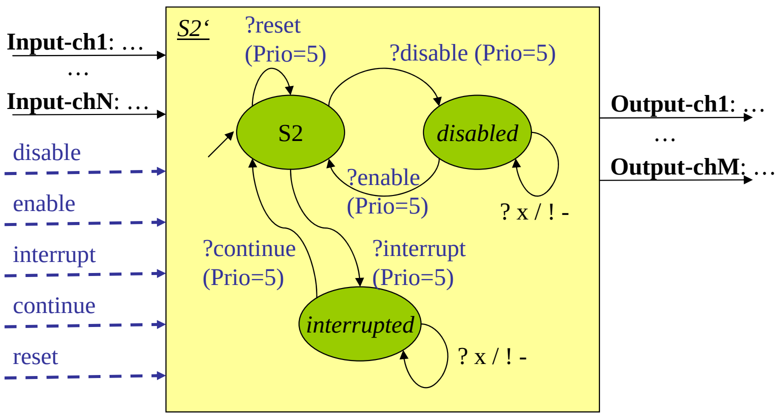

Figure 4.29.: Realization of all basic service relationships (SSD + STD)

have unique names, it might be necessary to make use of further identifiers (e. g. S1_S2_1).[21]

Realization of complex horizontal service relationships (relationship services)

So far we have shown how the basic service relationships can be realized. Now we will have a closer look at the complex horizontal service relationships. As mentioned in 4.6 (*Identification of service relationships*) the horizontal service relationships can be put down to the basic service relationships. The transformation of a horizontal relationship into basic relationships sometimes can be complex. In our approach, the logic behind such a transformation is "hidden" in a service, too. For example, for the horizontal service relationship XOR, we introduce an XOR service that is responsible for realizing the relationship, i. e. sending the actions ENABLE and DISABLE to the services which mutually exclude each other. We call such a service - which is responsible for realizing a complex service relationship - a *relationship service*. (See Section 4.8.4, *Application to the case study* for examples.)

Relationship services are special services. They can be considered to be pieces of functionality which are observable at the system boundaries. However, the relationship services can not be triggered by a user. They merely realize complex horizontal service relationships.

When taking a closer look at the horizontal service relationships of our running example, we see that some relationships depend on the *execution status* of one or more services. For example, the horizontal service relationship MAX2PAR indicates that at most two of the adjustment directions can be carried out simultaneously. As a consequence, the service which realizes the MAX2PAR relationship has to "know" the execution status of the adjustment services. To be more concrete, the MAX2PAR service has to know which one of the adjustment service is currently running, i. e. which service is currently ACTIVE or INACTIVE.

[21]As atomic and combined services should have unique names too, this is automatically achieved. However, in our methodology we will also introduce further services (see below) which could have the same name.

Providing information about the execution status of a service Similarly to the realization of the basic service relationships, we modify the service specifications to provide information about their execution status. Figure 4.30 contains the schema for modifying a modular service specification (i. e. the core behavior which might already have been adopted to handle basic service relationships). The syntactic output interface is enlarged by an output channel (depicted by a dotted arrow in the graphic) and the logical output actions ACTIVE and INACTIVE. Two high-level states are introduced: INACTIVE and ACTIVE. Per default the service is in the INACTIVE state. On receiving the first input action which triggers the service execution (STARTACT in the graphic), it sends the output action ACTIVE to a service realizing a service relationship which depends on the execution status. Within the state ACTIVE the service is executed according to its modular specification (according to the core behavior with might already have been adopted to handle basic service relationships). When putting out the last output action (ENDACT in the graphic), the service leaves the ACTIVE state and sends the action INACTIVE.

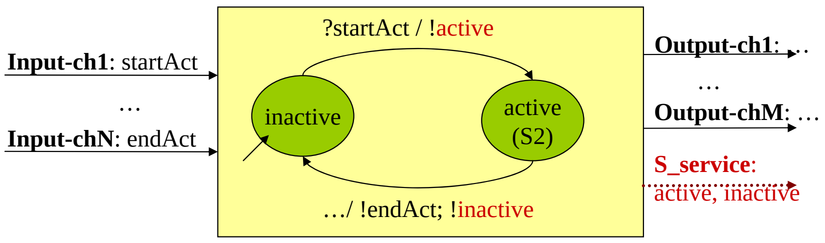

Figure 4.30.: Providing information about the execution status of a service (SSD + STD)

As the actions ACTIVE and INACTIVE introduced above have a special role in our methodology - as the basic actions (see above) - we give them a special name: *status actions*. The channels on which status actions are carried are formated differently, namely by dotted arrows. However, as mentioned in Section 3.1 (*System Structure Diagrams (SSDs)*) they have the same semantics as dashed or solid channels.

In Section 4.5.1 (*Concepts*) we introduced a channel for each input and output action, respectively, although the notational techniques allow the sending of multiple actions on one channel. The motivation for this is the following: If a service is connected to an input channel it receives all the input actions. However, as services are partial behaviors, a service does not have to handle each action. To keep the model simple we thus only sent those actions to/from a service which it needs for its execution. However, as far as inner channels (transmitting basic actions and status actions) are concerned, we allow multiple actions to be sent on one channel. This makes sense as basic actions and status actions are specifically introduced for a service.

During the examination of two case studies, we only identified horizontal service relationships that needed the information whether a service was active or inactive. However, other execution stati could be necessary for the realization of a horizontal service relationship, too. For example, if a service is in a stand-by mode. In that case, the modular service specification would have to be modified accordingly to the schema presented above. The syntactic output interface would be enlarged by the action STAND-BY and three high-level states would be introduced. As we can see, our schema can be adopted for an arbitrary set of executions stati.

Figure 4.31 summarizes the concepts introduced above: In general, if we want to combine two services S1 and S2 being related to each other by a horizontal service relationship (<horRel>), we obtain the model of Figure 4.31. The service <horRel> which is responsible for taking care of the realization of the horizontal relationship controls the execution of the services S1 and S1 by sending the basic actions (ENABLE, DISABLE, INTERRUPT, CONTINUE, and/or RESET). As the relationship <horRel> may depend on the execution status of S1 and/or S2, the services S1 and S2 send the actions ACTIVE and INACTIVE to the <horRel> service.

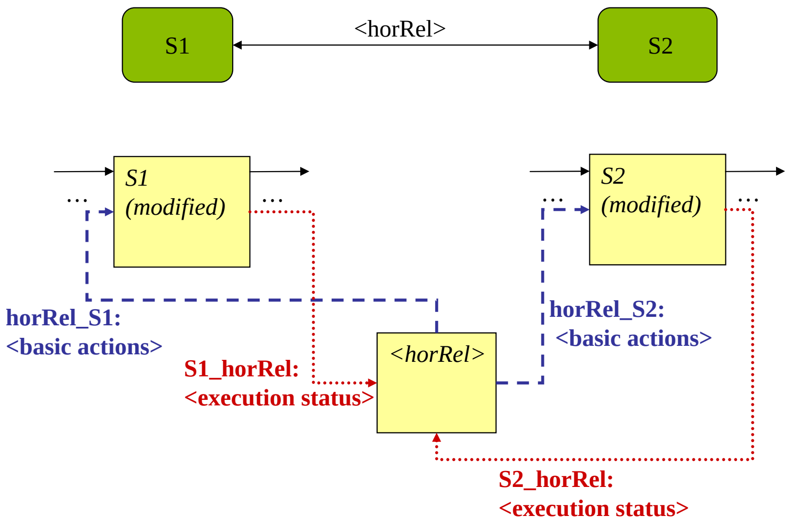

Figure 4.31.: Interplay of services according to horizontal service relationship (SSD)

In order to keep the model simple, we only modify the modular service specifications if necessary. If the horizontal relationship <horRel> of Figure 4.31 does not depend on the execution status of S1 and S2, the channels S1_HORREL and S2_HORREL are not introduced.

Often, services have to be modified because of service relationships pointing at their parent (or the parent of the parent, ...). In that case the service specification has to be modified accordingly later. We will come back to this point with an example in Section 4.8.4 (*Application to the case study*).

Horizontal service relationships pointing at super services (status calculating services and basic handling services) As explained at the beginning of this section, our notion of service is scalable. During the combination process we combine services (based on the service relationships between them) and get more comprehensive services. For example the service MANUAL ADJUSTMENT is hierarchically comprised of its sub services ADJUSTMENT OF BACK, ADJUSTMENT OF DISTANCE, ADJUSTMENT OF REAR AREA, and ADJUSTMENT OF FRONT AREA. If a horizontal relationships points to a hierarchically decomposed services, e. g. the service FRONT DOOR OPEN points to MANUAL ADJUSTMENT, the hierarchical service has to be able to handle the basic actions. What it means if a hierarchical service is affected, enabled or disabled for example, has to be designed individually. For example, it could mean that all sub services are disabled or just a dedicated one. The same holds for the execution status. A hierarchical service might have the status RUNNING if one of its sub services is running or if all are running in parallel. In general, the affection and the execution status of a hierarchical service have to be determined individually based on its sub services and according to the requirements (See Section 4.8.4, *Application to the case study* for examples).

Additional services have to be introduced which calculate the execution status of a combined service as the execution status of a combined service in general depends on the execution stati of its sub services. We call these services (which calculate the execution status of combined services *status calculating services*. See Section 4.8.4 (*Application to the case study*) for several examples.

Moreover, if a combined service has to handle basic actions (because of horizontal service relationships pointing at it) it has to be specified how this is realized. In general, its sub services have to handle these basic actions. We introduce so-called *basic handling services* which delegate the handling of basic actions (which a combined service receives) to the sub services of the combined service. Again see Section 4.8.4 (*Application to the case study*) for several examples.

By using the schema presented above, the modular service specifications are adopted (to realize the service relationships) step by step.

Often, a service has more than one service relationship pointing at it. In our running example for instance, both the services FRONT DOOR OPEN and LOW BATTERY have an effect (ENABLE and DISABLE relationship) on the service MANUAL ADJUSTMENT. These effects might be conflicting. For example, the FRONT DOOR OPEN service could demand the enabling of the MANUAL ADJUSTMENT service while the LOW BATTERY service might demand the disabling. It has to be decided how to proceed in this case. The effect on the MANUAL ADJUSTMENT service has to be calculated based on the effects that the FRONT DOOR OPEN and LOW BATTERY services have on it. Again, see Section 4.8.4, *Application to the case study* for examples.

Remarks on the scoping of variables

In Section 4.6 (*Identification of service relationships*) we assigned persistent data (variables) that several services make use of to the least common parent in the service hierarchy. As in our methodology, services are hierarchically combined resulting in hierarchical SSDs, we get the following consequence: Variables which are assigned to inner nodes of the service hierarchy (i. e. the least common parent service) are assigned to the respective, hierarchically decomposed SSD. In Section 4.6 (*Identification of service relationships*) we also said that variables which are assigned to a service are in the scope of the service itself and all its sub services. This matches to the semantics of our notational techniques (see Section 3.2.4, *Semantics of the combination of STDs and SSDs*).

Conflicts due to services writing on the same output channel

As already depicted in Figure 4.1, the case might occur in which services write different values on the same output channel. This might lead to conflicts. Also, one service could demand that an action is not written in a specific time interval while another service wants to send the action.

In order to handle these conflicts, conflict solving services also have to be introduced if services share output channels.

Conflicts due to services writing the same variable at the same time

As mentioned several times before, services might operate on the same persistent data. For example the user management (which is not part of our running example) saves the seat position according to the four dimensions (back, distance, height front, and height rear). The manual and the automatic adjustment services read this data. Allowing services to share the same data might lead to conflicts. The cases in which services only read the same data, or one service writes the data and several other services read it (like in the just mentioned example), we do not have a conflict. However, in general the case might occur in which several services write different values to a variable at the same time.

These conflicts lead to unwanted partiality. Allowing several services to access - to be more precise: to *write* - the same data at the same time, we do not have a constructive semantics anymore. Although the model allows the concurrent writing of the same variable this can not be implemented in case of conflicts.

In our case, such conflicts lead to contradictory or missing requirements. From a users' perspective, it should not be possible to write different values at the same time. Either the requirements are contradictory or a requirement (e. g. determining a priority order) is missing. Therefore, we enrich this step of the methodology by explicitly solving these conflicts: During the combination process, conflicts due to concurrent writing of shared variables have to be detected. This can be done schematically. Each time we reach an inner service which has data assigned, we go through all the services of its subtree and identify conflicts. This is eased by the data dependency relationships within the service graph. The conflicts then have to be solved specifically.

Further remarks

Before the combination of the the modular service specifications we had a *local view* on the functionalities as we modeled the single services separately. Consequently, after the combination process, states of different services might have the same name. Therefore, if unique state names are demanded, states might have to be renamed during the combination process.

In Section 4.4 (*Identification of atomic services*) we gave a rule of thumb what atomic services are. We informally defined them as "smallest" pieces of (partial) black box functionality which can be accessed/observed/distinguished by a user and which are likely to be reused. Based on the concepts introduced in this section, we modify this informal definition. For example, if a service relationship only refers to a part of the service behavior, it might be appropriate to split the service behavior into sub services and let the service relationship point to the sub service. Also, if the execution status of a service is determined due to the execution stati of its sub behaviors, it might also be necessary to introduce sub services.

In our methodology we make use of persistent data to allow services to operate on the same data. If data is assigned to a service, it is in the scope of all of its sub services. Mapping our logical architecture which abstracts from distribution (see Section 1.2, *Content of this thesis*) to a distributed architecture, we have to be aware of the following fact: Services which share data and which are mapped to different components can not communicate via shared variables anymore. Consequently, this dependency has to be solved by assigning the data to a service and having it requested by others. Note that this fact might be a criteria for a specific mapping.

4.8.2. Notational technique(s)

As the combination of services results in services again (our notion of service is relative), we again make use of (hierarchical) SSDs and STDs for the specification of combined services.

Remark: As mentioned above, we chose automata for the modeling of the service behavior. Of course, other notational techniques are suitable as well. However, if other notational techniques are chosen, the introduced concepts have to be defined on them.

4.8.3. Methodological steps

Figure 4.32 summarizes the combination of the (formal) modular service specifications graphically:

In the first activity, the services which are to be combined (next) are identified. Note that as mentioned above, we follow a bottom-up procedure (concerning the service graph) for the combination process. We start with the combination of the atomic services and go up until we obtain the functionality which is represented by the root of our service graph.

The next activity depends on the relationship(s) between the services to be combined: basic relationship or (complex) horizontal relationship. In case we have a **(complex) horizontal relationship** between them (right part of the activity digram of Figure 4.32) we have to perform the following steps: a service realizing the logic behind the horizontal relationship (e. g. XOR) is introduced. Additionally, the horizontal service relationship is broken down into basic relationships. Afterward we proceed for each basic relationship as described in the following paragraph. Furthermore, if need be, a conflict solving service is introduced.

If we have a **basic relationship** between the services, we perform the following steps: First, we introduce a channel between the influencing and the influenced service. The channel is directed and starts at the influencing service and leads to the influenced service. Note that the influencing service can be the service which realizes a horizontal relationship or another service which is the result of another service combination. As naming convention we suggested to name a channel S1_S2 if the channel connects service S1 with service S2. Accordingly to the newly introduced channel, the syntactic and semantic interfaces of the influencing and the influenced service have to be modified, respectively. These activities are refined in Figure 4.33.

As far as the **syntactic interface** of the **influencing service** is concerned, output channels (containing the basic actions sent to the influenced service) are introduced (starting at the influencing service and leading to each influenced service, respectively) and - if need be - input channels (containing the status actions sent by the influenced services) are introduced (starting from the influenced services, respectively, and leading to the influencing service. It has to be determined which basic actions have to be sent on the output channels in order to influence the behavior of the influenced services. This depends on the relationship which is realized. For example if the ENABLE and DISABLE relationship is realized, the actions ENABLE and DISABLE are sent on it. Additionally, it has to be determined which basic actions are received by the influenced services.

The **semantic interface** of the **influencing service** is modified in the following way: It has to be specified at what point in the service execution of the influencing service, the basic actions have to be sent to the influenced services. As mentioned above, this may depend on the execution status of the influenced services.

The **syntactic interface** of each **influenced services** has to be modified complementarity. An input channel is introduced containing the basic actions sent by the influencing service. Moreover, if need be, an output channel is introduced with the status actions that the influenced service needs.

Of course, the **semantic interface** of each **influenced services** has to be changed accordingly. The behavior has to be modified (according to the schemes introduced above) in order to handle the basic actions that can be received on the newly introduced channel. Furthermore, the execution status of the service has to be sent on the newly introduced output channel. Again this can be realized using the schemes presented above.

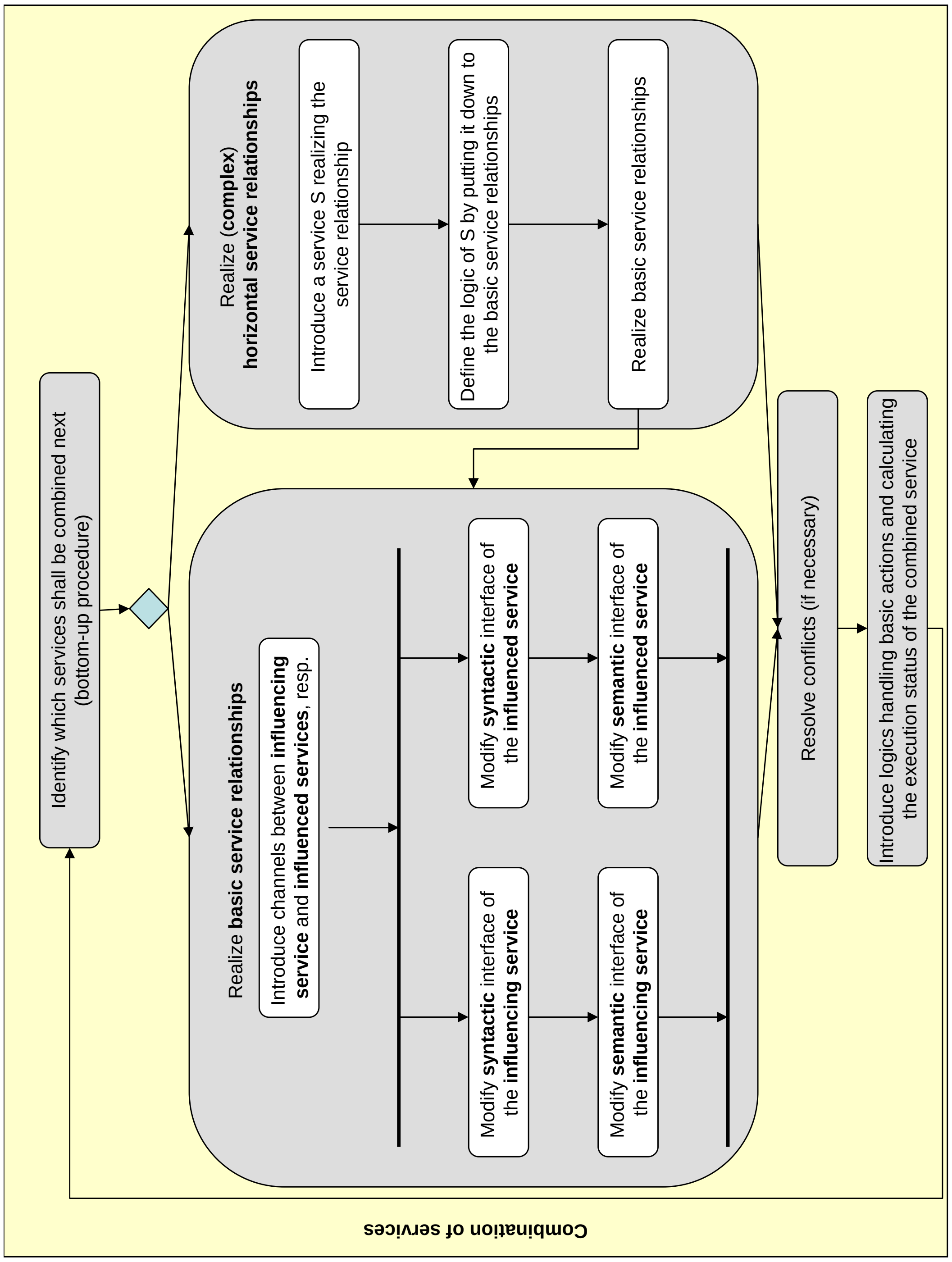
Combination of services
Identify which services shall be combined next (bottom-up procedure)
Realize (complex) horizontal service relationships
Introduce a service S realizing the service relationship
Define the logic of S by putting it down to the basic service relationships
Realize basic service relationships
Realize basic service relationships
Introduce channels between influencing service and influenced services, resp.
Modify syntactic interface of the influencing service
Modify semantic interface of the influencing service
Modify syntactic interface of the influenced service
Modify semantic interface of the influenced service
Resolve conflicts (if necessary)
Introduce logics handling basic actions and calculating the execution status of the combined service

Figure 4.32.: Combination of services on basis of service relationships (activity diagram)

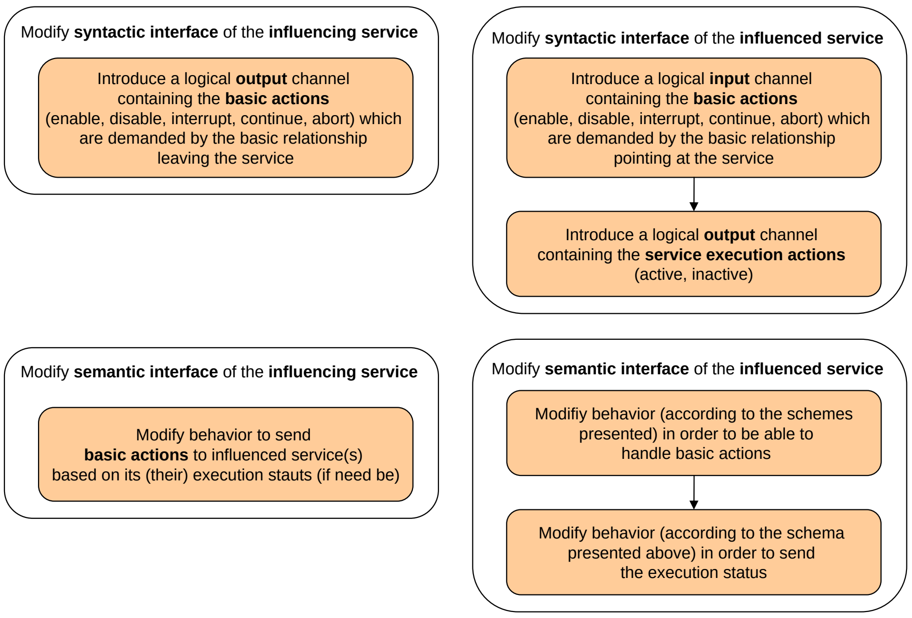
Modify **syntactic interface** of the **influencing service**
Introduce a logical **output** channel containing the **basic actions** (enable, disable, interrupt, continue, abort) which are demanded by the basic relationship leaving the service
Modify **syntactic interface** of the **influenced service**
Introduce a logical **input** channel containing the **basic actions** (enable, disable, interrupt, continue, abort) which are demanded by the basic relationship pointing at the service
Introduce a logical **output** channel containing the **service execution actions** (active, inactive)
Modify **semantic interface** of the **influencing service**
Modify behavior to send **basic actions** to influenced service(s) based on its (their) execution stauts (if need be)
Modify **semantic interface** of the **influenced service**
Modifiy behavior (according to the schemes presented) in order to be able to handle basic actions
Modify behavior (according to the schema presented above) in order to send the execution status

Figure 4.33.: Combination of services on basis of service relationships - Refined view (activity diagram)

After these activities, conflicts might be resolved. As described in Section 4.8.1 (*Concepts*) conflicts for example might be the result of services writing the same data at the same time. These conflicts can be identified with help of the data dependency relationship. Conflicts might also occur if different service write on the same output channels. Conflicts in general can not be solved according to a schema as they differ from case to case.

Services which are the combination of sub services may be affected by basic actions (e. g. the MANUAL ADJUSTMENT service is affected by the service FRONT DOOR OPEN), too. Therefore it has to be modeled what sub services (all or a subset) shall be affected in case a basic action occurs. The same holds for the execution status of a service which is combined of sub services. The execution status of the service combination has to be calculated on basis of the execution stati of its sub services. To that end, we introduce two more services: a *basic action logic* and a *status action logic*. (See proceeding paragraph for an example.)

4.8.4. Application to the case study

In this subsection we combine the services of the running example step by step (in a bottom-up manner, see Figure 4.13) until the overall system functionality is obtained.

Combination of the atomic manual adjustment services

First, the atomic services which are needed for the manual adjustment of the seat are combined. The formal specification of these services was already done in Section 4.7.4 (*Application to the case study*).

The combination of the ADJUSTMENT OF THE BACK BACKWARDS (ADJBACKBACKW) and the ADJUSTMENT OF THE BACK FORWARDS (ADJBACKFORW) services yields in the service ADJUSTMENT OF BACK. For the combination, the complex horizontal service relationship XOR has to be realized.

Figure 4.34 shows the result. The relationship service XOR is introduced which is responsible for realizing the service relationship. In case the ADJBACKFORW (ADJBACK´BACKW) service is active, it sends the action ACTIVE to the XOR service. On receiving this action, the XOR relationship disables the other service. As soon as the service execution is over, the ADJBACKFORW (ADJBACKBACKW) service sends the action INACTIVE to the XOR service which enables the other service again. In case both services are called at the same time, the XOR service disables the ADJBACKFORW service.[22]

[22]Note that this case is not covered by the requirements. We just chose one possibility.

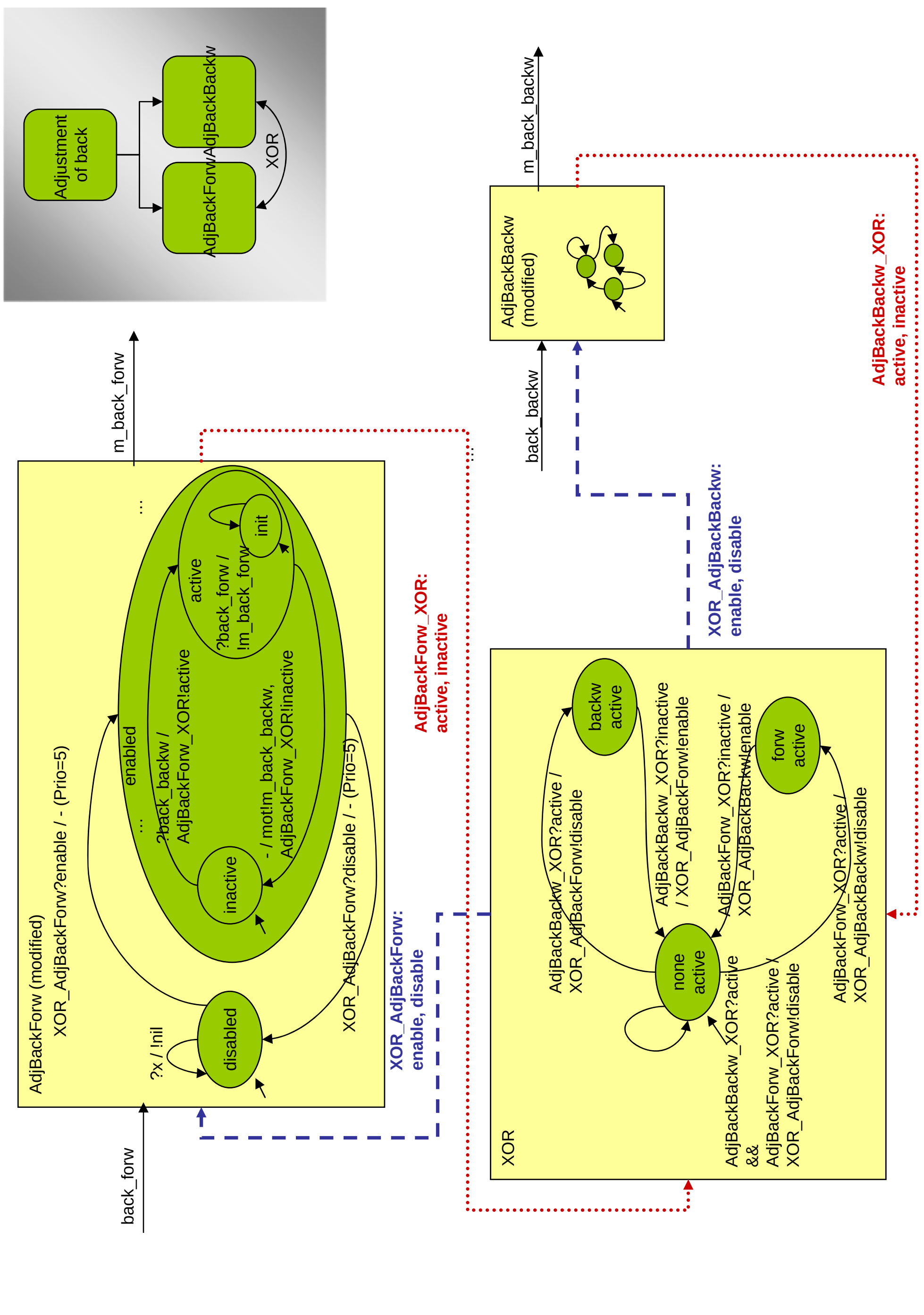
Adjustment of back
AdjBackForw
AdjBackBackw
XOR
AdjBackForw (modified)
XOR_AdjBackForw?enable / - (Prio=5)
back_forw
m_back_forw
?x / !nil
disabled
...
enabled
inactive
?back_backw / AdjBackForw_XOR!active
- / motlm_back_backw, AdjBackForw_XOR!inactive
active
?back_forw / !m_back_forw
init
...
XOR_AdjBackForw?disable / - (Prio=5)
XOR_AdjBackForw: enable, disable
AdjBackForw_XOR: active, inactive
...
XOR
AdjBackBackw_XOR?active / XOR_AdjBackForw!disable
backw active
AdjBackBackw_XOR?inactive / XOR_AdjBackForw!enable
none active
AdjBackForw_XOR?inactive / XOR_AdjBackBackw!enable
forw active
AdjBackBackw_XOR?active && AdjBackForw_XOR?active / XOR_AdjBackForw!disable
AdjBackForw_XOR?active / XOR_AdjBackBackw!disable
XOR_AdjBackBackw: enable, disable
AdjBackBackw (modified)
back_backw
m_back_backw
AdjBackBackw_XOR: active, inactive

Figure 4.34.: Combination of the services ADJUSTMENT OF THE BACK BACKWARDS and ADJUSTMENT OF THE BACK FORWARDS (SSD + extract of service graph)

The core behavior of each atomic service is modified: The syntactic interface is enlarged by channels receiving and sending the basic actions and actions about the execution service, respectively. The behavior (semantic interface) is modified accordingly (omitted for the service ADJBACKBACKW in the graphic). We hereby make use of the schemes presented above.

Figure 4.34 shows two problems of our methodology on which we will come back again in Section 7.2 (*Evaluation*): The services ADJUSTMENT OF THE BACK BACKWARDS and ADJUSTMENT OF THE BACK FORWARDS are specified independently of the fact how long the actual changing of the seat position (by the respective motor) takes place. The services send the status action INACTIVE in the next time tick after sending the output action. Either we have to assume that the changing of the seat position takes one time tick or we have to alter the model accordingly by taking into account the actual amount of ticks it takes to change the position. In the first case, the composition of the services leads to additional restrictions. In the second case we take into account restrictions given by the environment. Assuming the the changing of the seat position takes one time tick, the model of Figure 4.34 could be much simplified. For the combined service (ADJUSTMENT OF THE BACK) one SSD could be introduced. The corresponding STD would contain only one state with two transitions: ?BACK_BACKW / !M_BACK_BACKW and ?BACK_FORW / !M_BACK_FORW. Additionally, a priority ordering should be specified (PRIO=1) for the case in which both inputs are received at the same time. These two transitions exclude each other per definition. For this combination of two simple services, the composed model of our approach seems to be an overkill. In order to combine two more elaborate functionalities (which can not be represented by a single transition) by means of an XOR relationship, three states could be introduced: two (hierarchical) states realizing the behavior of the sub services, respectively, and one state in which both services are not active. Our methodology provides a systematic and schematic procedure how to combine services on basis of service relationships. However, for special cases like the XOR relationship, although our schema does result in a correct model, the model might be too complex. In this case one can restrain from the schema and make use of a simpler model. We will come back on this in Section 7.2 (*Evaluation*).

The combination of the other manual adjustment services is done analogously to the ADJUSTMENT OF BACK service and omitted here.

Combination of the adjustment services

Figure 4.35 depicts (a simplified model of) the MANUAL ADJUSTMENT service being the combination of its four sub services. The horizontal service relationship MAX2PAR controls the status of the four sub services ADJUSTMENT OF THE BACK, ADJUSTMENT OF THE DISTANCE, ADJUSTMENT OF THE REAR AREA, and ADJUSTMENT OF THE FRONT AREA. As soon as two services are active, the other two services are disabled. In case less than two services are currently active, the other services are enabled again.

Note, the MAX2PAR service needs the execution status of the services ADJUSTMENT

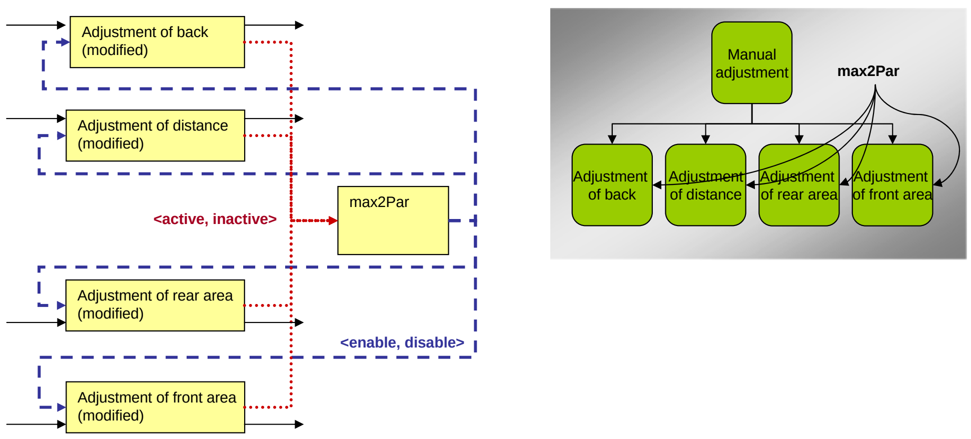

Figure 4.35.: Combination of the services ADJUSTMENT OF THE BACK, ADJUSTMENT OF THE DISTANCE, ADJUSTMENT OF THE REAR AREA, and ADJUSTMENT OF THE FRONT AREA (SSD + extract of service graph)

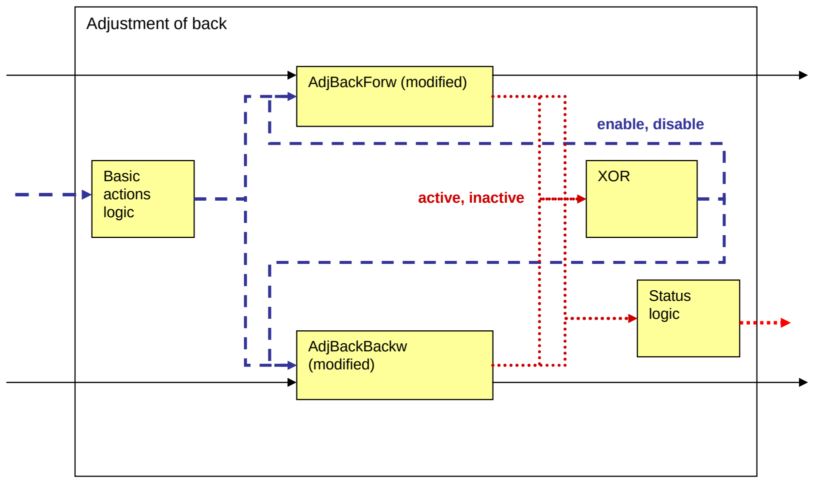

Figure 4.36.: Combination of the adjustment services - belated modification (SSD)

OF THE BACK, ADJUSTMENT OF THE DISTANCE, ADJUSTMENT OF THE REAR AREA, and ADJUSTMENT OF THE FRONT AREA. Also, it enables or disables these services. In the current version of the service specifications (as obtained in the previous paragraph), this information is not contained. Only their sub services (e. g. ADJBACKBACKW and ADJBACKFORW) can be enabled or disabled and provide their execution status. Therefore, the specifications of the four adjustment services have to be modified later. Figure 4.36 depicts the modified specification. Two more services are introduced. The BASIC ACTIONS LOGIC determines which sub services have to be enabled (disabled) if the combined service (ADJUSTMENT OF BACK) shall be enabled (disabled). We call this service a *basic handling service*. For example it would make sense to enable both sub services if the super-service has to be enabled. However, this depends on the context. Analogously, the service STATUS LOGIC calculates the execution of the combined service depending on the execution stati of its sub services. This also depends on the context. The service STATUS LOGIC is a *status calculating service*.

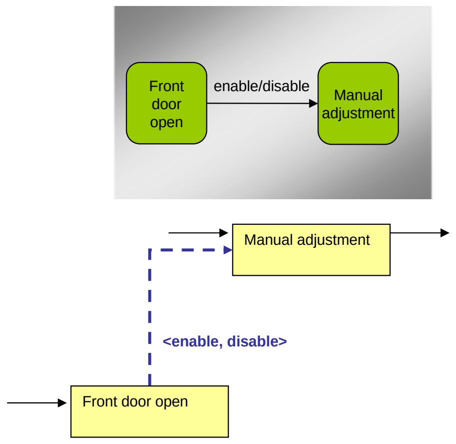

Figure 4.37.: Combination of the services FRONT DOOR OPEN and MANUAL ADJUSTMENT (SSD + extract of service graph)

Combination of the services FRONT DOOR OPEN and MANUAL ADJUSTMENT

Figure 4.37 contains the combination of the services FRONT DOOR OPEN and MANUAL ADJUSTMENT. In Section 4.7.4 (*Application to the case study*) we mentioned, that the FRONT DOOR OPEN service is trivial and the formal specification of its semantic interface might not make sense. However, now we see, that specifying this functionality as a separate behavior indeed makes sense. Its behavior is mainly given by the influence the FRONT DOOR OPEN service has on the MANUAL ADJUSTMENT service. Imagine the situation that the FRONT DOOR OPEN service might influence other services, too or could be in turn influenced by other services. Its behavior would become even more comprehensive in that case. Therefore, it makes sense to specify it modularly.

Note that the FRONT DOOR OPEN service does not depend on the execution status of the MANUAL ADJUSTMENT service. Therefore, the syntactic and semantic interface does not have to be enlarged in this regard.

Combination of the services FRONT DOOR OPEN, MANUAL ADJUSTMENT and ERR LOW BATTERY MANUAL

Next, we combine the services FRONT DOOR OPEN and MANUAL ADJUSTMENT (which we already combined in the previous paragraph) with the service ERRLOWBATMAN (see Figure 4.38). On closer examination we realize a problem: The situation could occur in which the MANUAL ADJUSTMENT service is enabled by the FRONT DOOR OPEN service (because the front door is opened), but disabled by the ERRLOWBATMAN service (because the battery is too low). The enabling by the FRONT DOOR OPEN service could "override" the disabling by the ERRLOWBATMAN service. This sort of problem is referred to in literature under the term *feature interaction* (see [Zave, 2001] for instance).

In order to handle this problem, we modify our model as shown in Figure 4.39. We introduce a special service called CONFLICT SOLVER which handles the basic actions

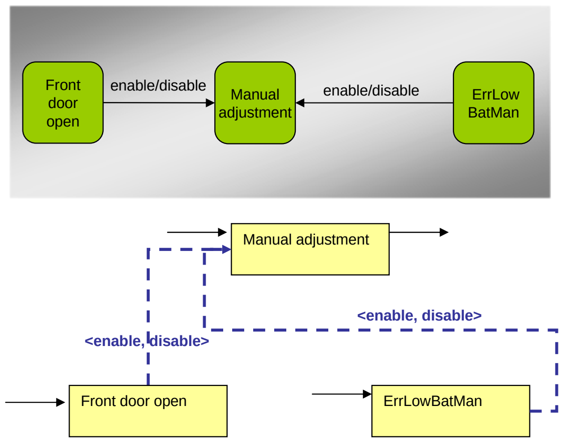

Figure 4.38.: Combination of the services FRONT DOOR OPEN, MANUAL ADJUSTMENT, and ERR LOW BATTERY MANUAL without conflict solving (SSD + extract of service graph)

sent by the services FRONT DOOR OPEN and ERRLOWBATMAN. Only if both services (FRONT DOOR OPEN and ERRLOWBATMAN) enable the service MANUAL ADJUSTMENT, it is really enabled. Due to reasons of clarity, we named the channels a,b, and c instead of using the (longer) name of the channels. Furthermore, some of the transition labels are committed. They can be obtained analogously to the labels displayed.

The conflict solving service has four control states. In state BOTH ENABLING both services (FRONT DOOR OPEN and ERRLOWBATMAN) want to enable the MANUAL ADJUSTMENT services. In the state BOTH DISABLING both services want to disable them. In the other states, exactly one of the services wants to enable the MANUAL ADJUSTMENT.

Combination of the services FRONT DOOR OPEN, MANUAL ADJUSTMENT, ERR LOW BATTERY MANUAL, ADJUSTMENT BY MEMORY and ERR HIGH VELOCITY

Next, we want to combine the services FRONT DOOR OPEN, MANUAL ADJUSTMENT, ERR LOW BATTERY MANUAL, and ADJUSTMENT BY MEMORY. Figure 4.40 shows the result. Basically, the XOR relationship between the service MANUAL ADJUSTMENT and the service ADJUSTMENT BY MEMORY has to be realized. As this relationship depends on the execution status of both services, MANUAL ADJUSTMENT and ADJUSTMENT BY MEMORY send this information to the relationship service XOR which is responsible for realizing the relationship. According to this status information, the XOR service enables or disables the services. Note that as the service ADJUSTMENT BY MEMORY can also be enabled or disabled by the service ERR HIGH VELOCITY, we again introduce a conflict solving service (CONFLICT SOLVER2). The conflict solver of Figure 4.39 was modified to also handle the enable or disable actions of the service XOR.

Taking a closer look at this example, we see the main advantage of our approach. We do not have to know the white box specification (STD) of the service ADJUSTMENT BY MEMORY. This functionality can be developed in isolation. We just demand the

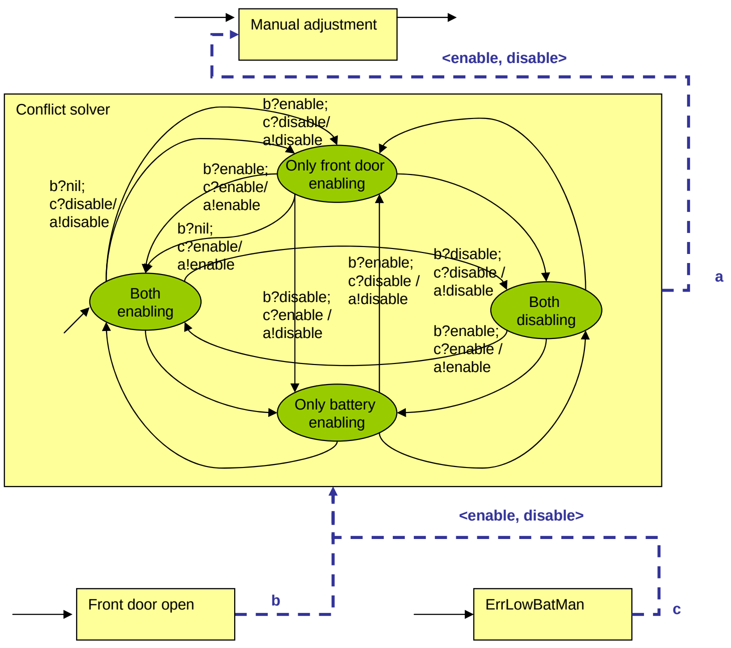

Figure 4.39.: Combination of the services FRONT DOOR OPEN, MANUAL ADJUSTMENT, and ERR LOW BATTERY MANUAL with conflict solving (SSD)

requirement that ADJUSTMENT BY MEMORY has to send the status actions ACTIVE and INACTIVE in order to submit its execution status to the service XOR. Additionally, we demand the requirement that ADJUSTMENT BY MEMORY must be able to handle the basic actions ENABLE and DISABLE which are sent by the XOR service. The realization of these requirements again can be achieved by implementing the standard control interfaces.

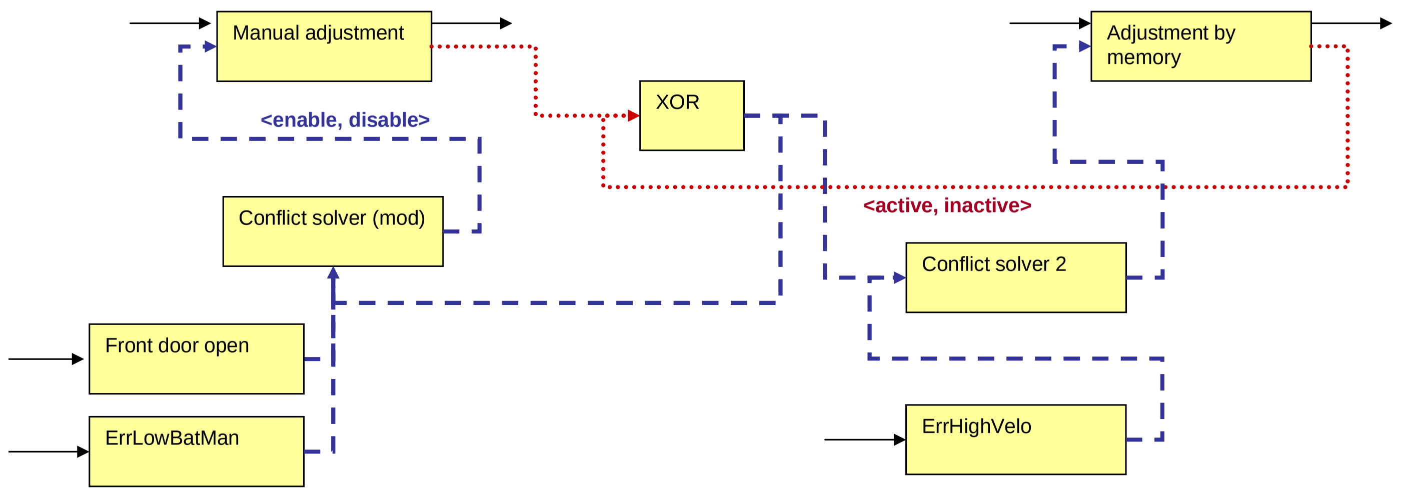

Figure 4.40.: Combination of the services FRONT DOOR OPEN, MANUAL ADJUSTMENT, ERR LOW BATTERY MANUAL, and ADJUSTMENT BY MEMORY (SSD)

The memory services make use of commonly shared persistent data. In this case we have to investigate if conflicts might occur. This would be the case if a service would write the same variable at the same time. However in our example, the services only read the same data. Thus no conflicts can occur.

Combination of all subservices

The last step is to combine all services in order to obtain the overall system functionality. At the end of this step we get the result of our methodology. In the proceeding section, the overall model is shown.

4.9. Result

Figure 4.41 contains a simplified model of the overall system functionality. At the end of the requirements engineering phase it is important that the functional requirements are captured *completely* and *consistently* and that they are *described precisely*. The precise meaning of the functionality is given by the formal foundation of our notational techniques. We showed how our model-based requirements engineering approach helps at identifying missing requirements. To validate the requirements, the model can be simulated. However, according to the semantics of underspecified transitions (as defined in Section 3.2.3, *Transitions*), our notational techniques do not have a constructive semantics. In order to simulate the model, a tool could choose one of the possible outputs nondeterministically for each channel to deal with this situation.

To finally make the model total, each automaton now has to be made total (if not already been done) to specify the usage behavior completely. Basically, we face three different forms of canonical completions to extend a partial service specification to a total service specification [Schätz and Salzmann, 2003]:

- Chaotic completion: In this case arbitrary behavior is possible on receiving underspecified input. (The behavior can not be distinguished from nondeterministic behavior if underspecified input is received.)
- Operational completion: Empty output is produced in case the service exhibits undefined behavior at some point of time.
- Error completions: The service produces an error message if it exhibits undefined behavior.

From a requirements engineering point of view, a canonical completion does not make sense. If there exist situations in which it is not clear what a service (or the combination of services) has to do, missing requirements are the reason. Consequently, further functional requirements have to be elicited.

As far as our running example is concerned, we omit the step of making the specification total.

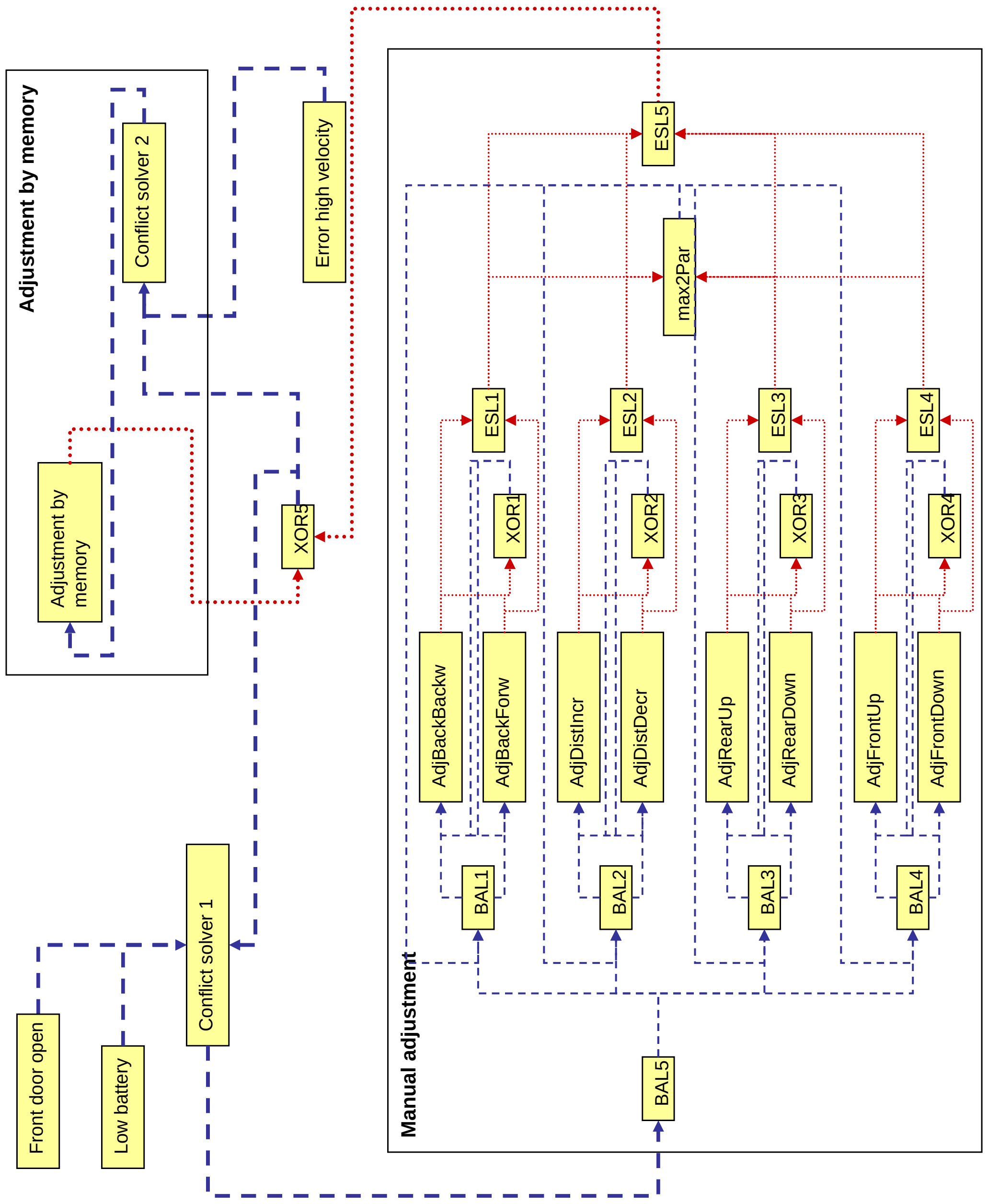

Figure 4.41.: Combination of all services (SSD)

The formal model of the overall system functionality is not the only result of our methodology. We also obtain the various artifacts that we produced during the process and which can be reused for the specification of future systems:

- Informal (textual) service specifications of
  - atomic services
  - combined services
- Specification of logical input and output actions
- Specification of logical input and output channels
- (Informal and formal) Specification of complex horizontal service relationships
- Formal service specifications of
  - atomic services
  - combined services

4.10. Further considerations

In this section we make some considerations about our approach which can not be allocated to one of the previous sections.

4.10.1. Views onto the service graph

When specifying the functionality of medium to large scale systems, hundreds of (atomic and combined) services can be identified. Consequently, the service graph can be quite big and thus difficult to overview. In order to solve this problem, it is usually very helpful to reason about different aspects in isolation. Different *views* onto the system provide an extract of the service graph. They display one aspect at a time. In the following we give ideas what kind of views onto the service graph of our methodology could be useful to maintain an overview.

One possibility is to only look at *sub graphs* of the service graph. These sub graphs could be called logical (functional) sub systems. For example, the services of the power seat control system are a sub system of a bigger system. Due to the modular character of our approach, logical sub systems can be reasoned about in isolation and combined later.

Another possibility to introduce views onto the service graph is to only look at a *subset of the services* according to service types. For example, only the conflict services could be looked at to get an understanding about possible conflict situations. If we go one step further, additional service types could be introduced, e. g. error services, initialization services, configuration services. Services could be also classified according to different behavioral patterns ("enabler"). Views could then be created containing just the services of one of these special service types.

We could also filter the information of the service graph according to the service relationships. For example, we could only display the services being connected by the ENABLE and DISABLE relationship. This might be helpful during design in order to identify the state space of the system.

If the overall development process provides a tracing model, we also have other criteria according to which views can be created. Concerning tracing we can distinguish between so-called *horizontal* and *vertical tracing*. Horizontal tracing relates artifacts within the same development phase whereas vertical tracing relates artifacts along the development process. The service relationships can be considered as horizontal tracing links for example. Vertical tracing links, e. g. between use cases and services or between services and functions (building blocks of the white box functionality of a system) could provide a basis for views as well.

4.10.2. Dependency analyses

The aim of our approach is the formalization of the functional requirements. As already mentioned several times, for multi-functional systems it is very important to make the various dependencies that exist between system functionalities explicit. The service graph is the first step toward this need as it visualizes the dependencies which are visible at the system boundaries.

However, the dependencies between the services can be investigated in more detail. To that end, we need a *calculus on service relationships*.[23] By means of such a calculus further service relationships could be calculated on basis of the already identified service relationships. Above others, inheritance relationships could be detected. For example the service ERRLOWBATMAN enables (disables) the service MANUAL ADJUSTMENT. This enabling (disabling) is realized by the enabling (disabling) of the sub services ADJUSTMENT OF BACK, ADJUSTMENT OF DISTANCE, ADJUSTMENT OF REAR AREA, and ADJUSTMENT OF FRONT AREA. In turn, these services realize the enabling (disabling) by forwarding the respective actions to their sub services. The ENABLE/DISABLE relationship thus is inherited to the sub services of the service MANUAL ADJUSTMENT.

However, this is only true as the basic handling services BAL1, BAL2, BAL3, BAL4, and BAL5 (see Figure 4.41) realize the enabling (disabling) of the manual adjustment service by enabling (disabling) the sub services, respectively. Therefore, the logic behind the basic handling services has to be taken into account when calculating the effects of the ENABLE/DISABLE relationship.

In order to calculate the effects of the MAX2PAR relationship, the logic behind the relationship service MAX2PAR (which realizes this relationship) has to be known, too.

Based on such a calculus, *dependency tables* could be introduced visualizing the dependencies between the services of a system. For example, it could be visualized which service is (indirectly) enabled by which other service or which service (indirectly) enables which other service.

[23]Currently such a calculus on service relationships is investigated in the PhD thesis of Johannes Grünbauer. It might also be appropriate for our service relationships.

4.10.3. Guidelines for the informal specification of functional requirements

In the methodology presented in this thesis, textually given functional requirements are turned into formal models step by step. Thus, the functional requirements which are given in prose, are the starting point of our methodology (see Section 4.3, *Starting point*).

Now that we have presented our methodology the question arises whether we can give some advice on how to better document the functional requirements. The better the functional requirements are given, the easier our methodology can be applied. The intention is to give guidelines for the informal description of the functional requirements so that they serve as a better starting point for our approach.

We suggest the following guidelines: The system boundaries have to be specified explicitly by listing all inputs and outputs.[24] Another guideline is that for each technical signal which is referred to in the requirements (e. g. the error message ERR_BAT_LOW_MAN) an explanation has to be given.[25] This helps in defining the logical input and output actions of our approach which abstract from the technical realization (concrete technical signals or messages). Furthermore, sub functionalities have to be given names. This simplifies the identification of services.

The functional requirements of our running example are already structured. For example Section 2.2 (*Requirements for the manual adjustment*) contains the requirements for the manual adjustment and Section 2.3 (*Requirements for the adjustment by memory*) contains the requirements for the adjustment by the memory functionality. This structuring can serve as a first clue for the structure of the service graph. Another way of preparing the functional requirements is to make use of the names of (both the basic and complex) horizontal service relationships. Due to the schemes for the realization of each basic service relationship, they are given a precise semantics. This is important as it is often unclear what the exact relationship between functionalities is.

Furthermore, when making use of the horizontal service relationships already during the informal specification of the functional requirements, the values of the respective parameters have to be given, too. This helps in detecting necessary information which otherwise would have to be elicited later. As the semantics of the horizontal service relationships are given (by the schemes for the standard control interfaces) the habitual language use in informally specified requirements is stricter. For example, the requirements engineer has to think more carefully about using the term "reset" or "disable".

If we carry these ideas to the extreme the following bold assumption arises: Do I need the plain text description of functional requirements at all? Or is it possible to specify the service graph (with the textual description of the atomic and combined services "attached" to the nodes) at once? If this is really possible should be investigated (see Section 7.3, *Outlook*). Although this seems to be a tempting idea, the main point

[24]This is already done in [Houdek and Paech, 2002] from which we adopted the functional requirements of our running example.

[25]Usually, in requirement specification way more technical signal names are used to describe the wanted system functionality.

of criticism lies in the fact that there is no 1:1 mapping between requirements and services. It demands real genuine work to identify the services. For our running example the mapping between functional requirements and services was quite easy. However for other systems this might be difficult. This definitely also depends on the system type of the system under specification. For business information systems (BIS) the functionality of which is characterized by larger *sequences of interactions* it might be error prone. Here, an overlapping of different use cases (making use of the same sub functionalities) should probably be done first to identify the services.

In this chapter, we introduced a methodology for modeling usage behavior of multi-functional systems. The basic idea of the approach is to combine modular service specifications on basis of their dependencies. To that end, we introduced a set of basic service relationships. For each basic service relationship we gave a standard control interface. By implementing these standard control interfaces, the modular service specifications (core behavior) can handle the service influences (service/feature interaction). In the following chapter (Chapter 5, *Extension of basic service relationships*), we take a closer look onto these basic service relationships, discuss, and extend them.

Chapter 5

Extension of basic service relationships

The main idea behind our approach is to combine modular service specifications on basis of their dependencies (service relationships). Hereby, the vertical service relationships determine the order in which the combination is done. The horizontal service relationships (both the complex and the basic ones) capture information about feature interaction (or "service interaction" in our terminology). Complex horizontal service relationships are broken down to basic service relationships (RESET, ENABLE, DISABLE, INTERRUPT, and CONTINUE). For each basic service relationship a *standard control interface* is given. Services which implement this standard control interface can handle the effects of the respective basic service relationship. The modular service specification thus is modified according to the standard control interfaces (see Section 4.8, *Combination of services on basis of the service relationships*).

In Section 4.6 (*Identification of service relationships*) we introduced a set of basic service relationships. This set of basic service relationships is sufficient to model our running example. However, for other systems these basic service relationships might not be powerful enough to express each possible service relationship.

For example, the introduced basic service relationships do not contain any information what has to be done with the service call in case a service call is made while the service is currently being disabled (or interrupted). Should the call be buffered and issued as soon as possible as the service is enabled again or should the call be ignored? Another functional requirement that can not be expressed by our basic service relationships is one demanding that an error message has to be issued in case a service call is made while a service is currently disabled (or interrupted).

In this chapter we extend the set of basic service relationships of Section 4.6 (*Identification of service relationships*). Due to this extension, our methodology becomes more powerful; however it also becomes more complicated.

In order to introduce the more powerful extended set of basic service relationships, another quite small case study is given (a memory cell). In Section 5.1 (*Requirements of another case study (memory cell)*) we give the informal (textual) functional requirements for this case study. In Section 5.2 (*Informal introduction*) we present the extended set of basic service relationships. The end of this chapter is given by a discussion on the extension of the set of basic service relationships (see Section 5.5, *Discussion*).

Contents

5.1. Requirements of another case study (memory cell)

In order to introduce the extended set of basic service relationships, we make use of another case study: a very simple version of an integer memory (i.e. a memory that stores integer values). In this section, we give the informal (textual), functional requirements for this case study.

For reasons of simplicity, we assume that the memory cells are given initial values (for example "0") and that the storage is limited to 20 values.[1]

1. It must be possible to turn the system on and off.
2. It must be possible to turn the system into a standby mode and a normal mode.
3. The system can only be turned into the standby mode, if the system is on (and not off).
4. If the system is turned on, it must turn to the normal (standby) mode if it was in the normal (standby) mode when it was turned off.
5. If the system is on, it is possible to write an integer value to a specified cell. (There are 20 cells altogether. Both the integer value to be stored and the ID of the cell have to be specified when a value shall be written to the memory.) After the successful writing of the value, an acknowledgment action should be issued ("writeOk").
6. If the system is on, it is possible to read an integer value of a specified cell.
7. If the system is off, no other functionalities except the turning on of the system can be called. This includes the switching to the normal (standby) mode and the read and write requests.
8. If the system is on, all other functionalities can be called.
9. If the system is turned off, currently served requests are aborted immediately.
10. If the system is turned on, the system enters the initial state.
11. If the system is off, no read or write requests are performed. If a read or write request occurs while the system is off, the request is ignored.

[1]The latter requirement is a system requirement and not a functional requirement. However, this restriction has effects on the functionality, namely that only 20 values can be stored.

12. If the system is switched to the standby mode, currently served requests are interrupted and continued when the system is turned back to the normal mode.
13. If the system is in the standby mode and a request occurs, the request is buffered and handled when the system is turned back to the normal mode. Additionally, an error message ("System standby mode. Request buffered.") is issued.

5.2. Informal introduction of the extended set of basic service relationships

In this section we will introduce the extended set of basic service relationships. First, basic considerations are made on how to systematically determine the set of basic service relationships (see Section 5.2.1, *Basic considerations*). Then, the actual service relationships are listed. Additionally, each service relationship is named (see Section 5.2.2, *Extension of the basic service relationships*).

5.2.1. Basic considerations

In our approach, we first specify the atomic services modularly, i.e. as if they were not influenced by other systems. Then, the modular service specifications are combined. During the combination process, service interaction has to be resolved. This is done by adapting the modular service specifications according to standard control interfaces.

In order to analyze the basic service relationships systematically, i.e. to analyze which basic effects a service relationship can have on a modular service specification, we look at the following questions:

1. How is the modular service specification *left*?
2. What is done *while* the modular service specification is left (i.e. disabled or interrupted)?
3. How is the modular service specification *resumed* again?

When a service forces the ending (interruption) of another service execution we face two possibilities: Either the service is aborted (interrupted) *immediately*, i.e. as soon as the action which forces the ending (interruption) arrives, or the services "is allowed" to proceed until a secure point is reached before it is (finally) aborted (interrupted). It may be questioned, if it is worth to distinguish between these two cases. It would also be possible to leave it to the service whether its execution should be stopped immediately or not. However, the distinction does make sense. For example, in some situations, a "secure stop" might be wanted (i.e. to save data first). In other situations, it might be inevitable to stop the service immediately, for example, when the airbag of a car is about to be fired and other services are stopped immediately to have more battery power. Therefore, we distinguish between the immediate stopping and the secure stopping of a service execution.

The next question is what shall be done if a service call is issued while the service is stopped (disabled or interrupted). Here, we again have two possibilities: The simple possibility is that service calls (or other data which are necessary for the service execution) are *ignored*. The more elaborate possibility is that service calls (or other data) are *buffered*. As soon as the service is enabled or continued again, the service can process the buffered service calls (data). Additionally, it can be specified whether or not an error message (error msg) shall be issued in case a service call is issued while the service is stopped.

Last, it has to be specified by the basic service relationship, how the modular service specification is resumed after it was left. The service execution could start at the *init* state again (i.e. it is enabled) or continue from the state the service was in when it was stopped (i.e. continued).

5.2.2. Extension of the basic service relationships

In the previous subsection, we identified alternatives how a service execution can be influenced. We investigated how a modular service specification can be left, what is done while the specification is left, and how the service specification can be resumed. All the possible and *sensible* combinations of these parts are listed in Table 5.1.

Table 5.1.: Extended set of basic service relationships

leave	reset	immediate		
	reset	secure		
while	disable/interrupt	immediate	buffering	error msg
	disable/interrupt	immediate	buffering	no error msg
	disable/interrupt	immediate	ignoring	error msg
	disable/interrupt	immediate	ignoring	no error msg
	disable/interrupt	secure	buffering	error msg
	disable/interrupt	secure	buffering	no error msg
	disable/interrupt	secure	ignoring	error msg
	disable/interrupt	secure	ignoring	no error msg
resume	enable			
	continue			

The combinations shown in Table 5.1 make up the extended set of basic service relationships which is a refinement of the set of basic service relationships which was already introduced in the previous chapter.

The first service relationship of the extended set of relationships is called *reset*. It is equivalent with disabling and immediately enabling a service. This case is addressed specifically, as it is not necessary to specify what the service has to do while it is being disabled. For example, as the service is enabled immediately after it has been disabled, no error message can be issued in case a service call occurs.

We can distinguish between the service relationships IMMEDIATE RESET and SECURE RESET. An IMMEDIATE RESET forces the service to immediately abort the service execution (and return in the initial state) whereas the SECURE RESET relationship allows the service to first proceed until a secure point is reached.

The other basic service relationships are self explaining. Each of these extended basic service relationships is realized by sending and receiving (handling) a logical action with the same name.

5.3. Standard control interfaces of the extended basic service relationships

During the combination phase of our methodology (see Section 4.8, *Combination of services on basis of the service relationships*), the formal modular service specifications are adapted to handle the effects of other services (which are captured by the horizontal service relationships). We introduced standard control interfaces describing how a formal modular service specification has to be adapted in order to handle these influences. In this subsection we give the standard control interfaces of the extended set of basic service relationships.

For each basic service relationship (of the extended set of basic service relationships) a standard control interface for its realization is given (see Sections 5.3.1, *Sub relationships of* RESET to 5.3.4, *Relationship* CONTINUE).

5.3.1. Sub relationships of RESET

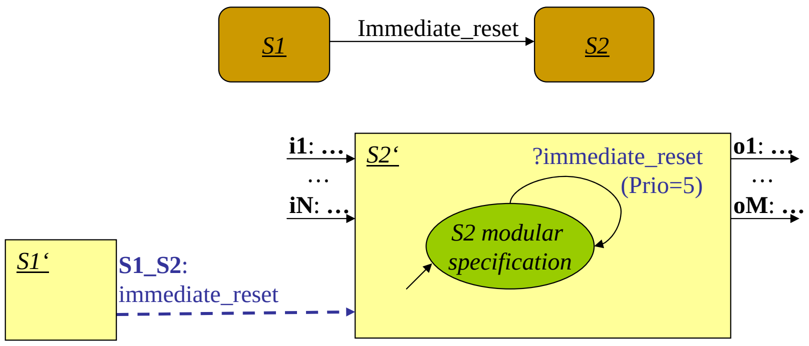

Figure 5.1.: Standard control interface of the basic service relationship IMMEDIATE RESET (SSD + STD)

Figure 5.1 contains the standard control interface of the basic service relationship IMMEDIATE RESET. In case an IMMEDIATE RESET command is received, the service S2 performs the higher level transition, i.e. the actual service execution is stopped and the initial state is entered.

The realization of the basic service relationship SECURE RESET is shown in Figure 5.2. A local variable (SEC) is introduced. On receiving the local action SECURE_RESET, the variable SEC is set to TRUE. The service specification is executed further until a secure point is reached. In the example shown in Figure 5.2, STATEN and STATEM are such (arbitrarily chosen) secure points. At such secure points, the transition {SEC==TRUE} - / - {SEC==FALSE} is fired, in case the variable SEC is set to TRUE. Thus, the service execution is reset. The variable SEC is set to FALSE. The secure points of a service execution are specific to the service and have to be determined for each service.

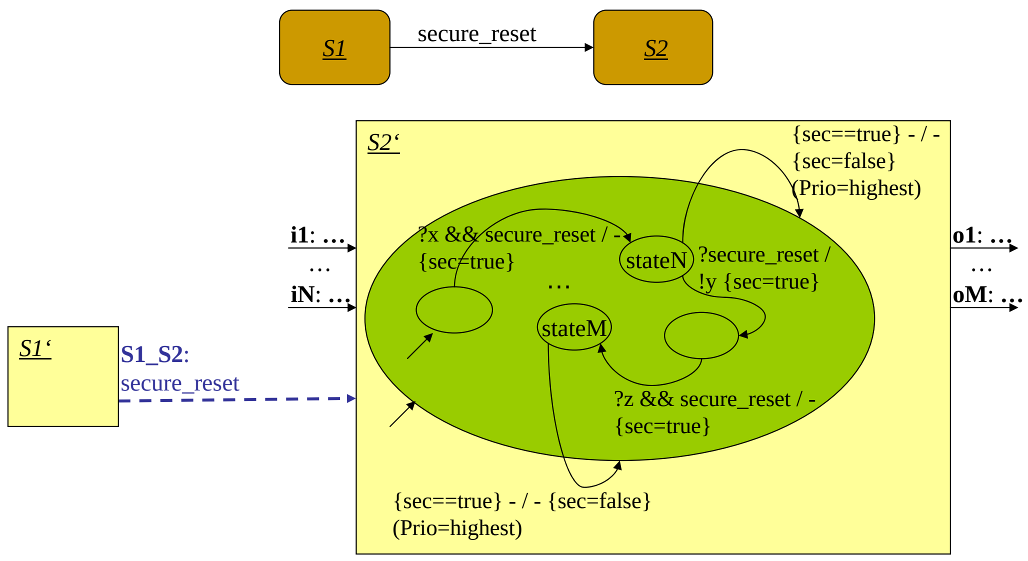
S1
secure_reset
S2
S2‘
{sec==true} - / - {sec=false} (Prio=highest)
i1: …
…
iN: …
?x && secure_reset / - {sec=true}
stateN
?secure_reset / !y {sec=true}
…
stateM
o1: …
…
oM: …
S1‘
S1_S2: secure_reset
?z && secure_reset / - {sec=true}
{sec==true} - / - {sec=false} (Prio=highest)

Figure 5.2.: Standard control interface of the basic service relationship SECURE RESET (SSD + STD)

5.3.2. Sub relationships of DISABLE/INTERRUPT

The standard control interfaces for the sub relationships of DISABLE/INTERRUPT (as noted before, DISABLE and INTERRUPT are semantically equivalent), are more complicated. We will introduce these sub relationships step by step.

Realization of the IMMEDIATE and SECURE part of the relationship

First we will present the standard control interface of the IMMEDIATE DISABLE (IMMEDIATE INTERRUPT) part of the relationships depicted in Figure 5.3. The standard control interface of the SECURE DISABLE (SECURE INTERRUPT) relationship can be obtained analogously to the SECURE RESET relationship (see Subsection 5.3.1, *Sub relationships of* RESET). The standard control interface for the realization of the IMMEDIATE INTERRUPT part can be obtained from Figure 5.3 by simply changing the logical actions and the respective stereotype IMMEDIATE DISABLE to IMMEDIATE INTERRUPT.

Realization of the BUFFERING and IGNORING part of the relationship

Figure 5.4 shows how the buffering of data can be realized. In the figure, the buffering and secure disable is shown. The other buffering variants (SECURE and/or INTERRUPT) can be obtained easily by using the concepts introduced above.

In order to realize the buffering, the modified service specification is divided into two sub services: a BUFFER and S2′. Latter is the modular service specification (S2) which has already been modified to handle the action DISABLE (see Figure 5.3). In the

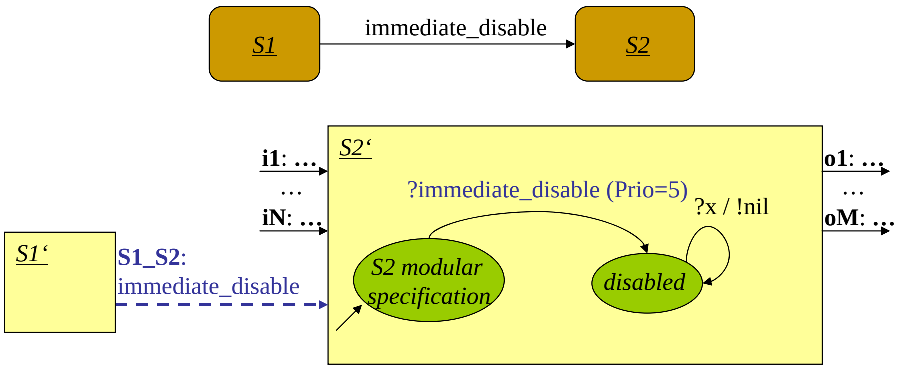

Figure 5.3.: Schema for the realization of IMMEDIATE DISABLE/INTERRUPT (SSD + STD)

NORMAL state, the buffer service behaves transparently[2]. It transmits the inputs to the modular service specification. The buffer service stores the logical input actions in a buffer variable "b".[3] Each input action "x" that is received is stored. The buffer service forwards the logical input actions (which are stored in the buffer variable b) in a FIFO (First-In-First-Out) fashion to the service S2′. In the BUFFERING state, the buffer service just stores incoming input actions in the variable "b" but does not forward it to S2′. Furthermore, it disables the service S2′ on entering the BUFFERING STATE.

The buffer variable "b" that we use to store data while a service is disabled (interrupted) is local to the service. ˆ is used to represent the concatenation of both characters and strings. | | depicts the logical OR.

Another important point to mention is the following: We can decide between services for which the service call is the only input (i.e. no further input data is needed for the service execution) and services that need additional input data for the service execution. By "service call" we denote the very first input action by which a service is triggered (if a service needs an input action to start its execution). In case a service is BUFFERING DISABLED which does need additional data this data is stored/buffered. When the service is enabled and called again, these buffered data is (still) at the beginning of the queue "b". Thus, a BUFFERING DISABLE does not make sense for this kind of service. However, it does make sense for services for which the only required input data is the service call.[4] In case, additional data is buffered, this has to be deleted before the enabling of the service. Of course, it is also possible to specify the buffer service accordingly, i.e. to only store the data after the next incoming service call (including the service call of course).

Figure 5.5 contains the standard control interface for the realization of the IGNORING DISABLE part of the basic service relationships. Again the IGNORING INTERRUPT interface part can be obtained by just changing the actions accordingly.

[2]modulo time delay

[3]Note that for reasons of clarity we only make use of one variable here. For each input channel, a variable has to be introduced.

[4]Note, that a BUFFERING INTERRUPT always makes sense as the service is continued (after the interrupt) at the place in the execution flow it was interrupted an thus needs the buffered data in any case.

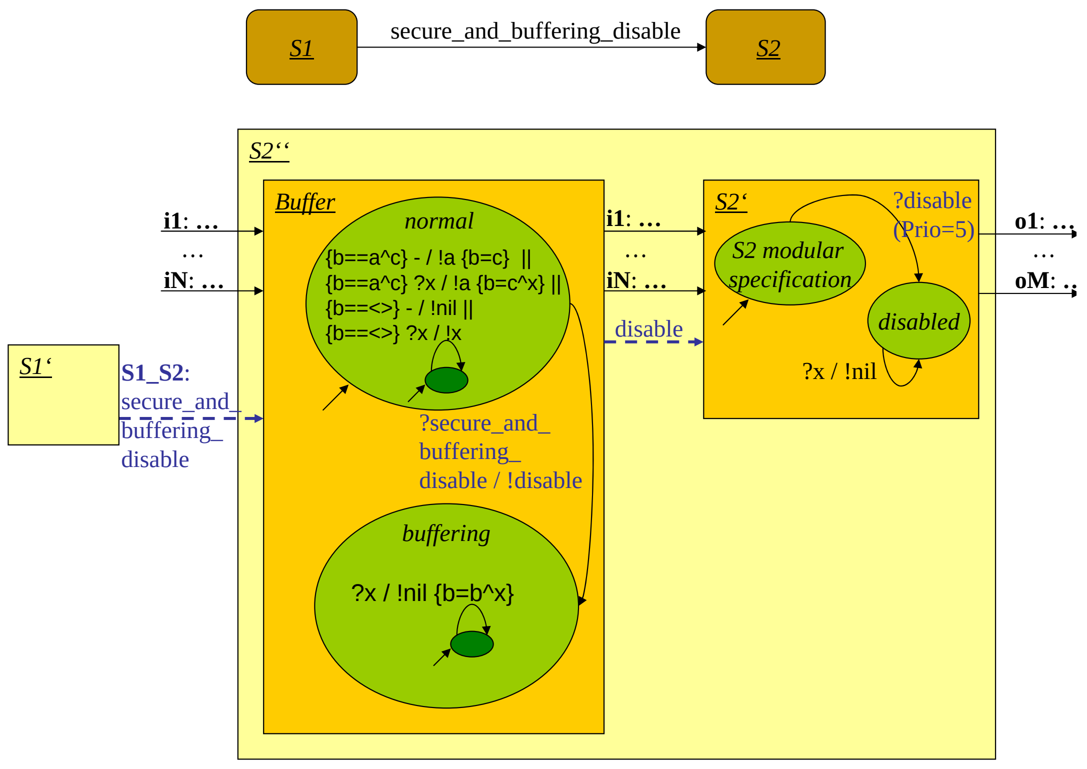

Figure 5.4.: Standard control interface for BUFFERING (SSD + STD)

It might not be clear why we need the "fake" buffer in the NORMAL state of the buffer service that immediately forwards the data. As motivation, imagine the following situation. A service is BUFFERING INTERRUPTED and (a little bit later) CONTINUED again at the same point in the service execution where it has been interrupted. After it is continued, the service first processes the buffered data. However, while processing the buffered data, new data might already come in. This input data then is buffered until the previously buffered data has been processed. Additionally, this construction is a way to handle time delays in case a service call is received while the service is currently active (and thus not in the initial state).

Realization of the ERROR MSG and NO ERROR MSG part of the relationship

The realization of the error message part is quite easy. Figure 5.6 shows the standard control interface for the relationship IMMEDIATE AND IGNORING DISABLE WITH ERROR MESSAGE (X). On receiving a service call in the IGNORING state, an error message is sent. The opposite case, in which no error message is sent while the service is disabled, can be obtained trivially by leaving away this output.

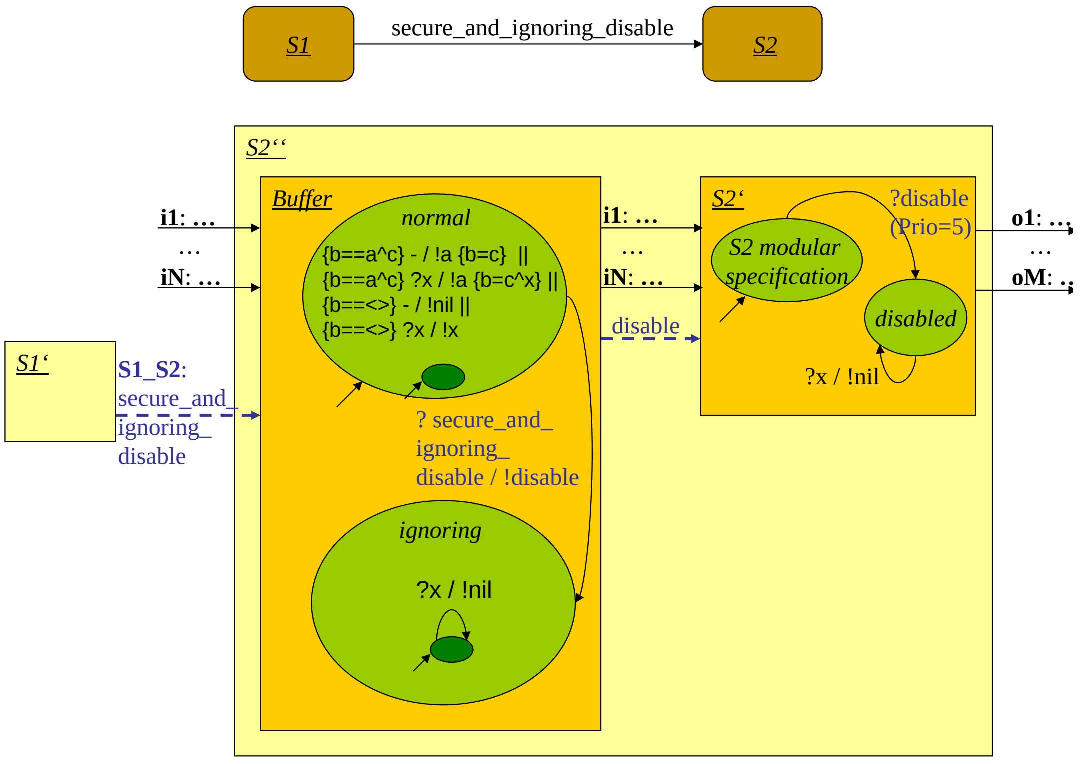

Figure 5.5.: Standard control interface of the IGNORING DISABLE part (SSD + STD)

Final standard control interfaces of the basic service relationships DISABLE and INTERRUPT

All sub relationships of the DISABLE/INTERRUPT relationship can be obtained by combining the schemas presented above.

Practical examples for the application of the more complex basic service relationships can be found in Section 5.4.4 (*Combination of services on basis of the extended set of basic service relationships*).

5.3.3. Relationship ENABLE

The realization of the basic service relationship ENABLE stays the same (see Section 4.8, *Combination of services on basis of the service relationships*) and is shown in Figure 5.7. On receiving the action ENABLE the service leaves the DISABLED state and enters the NORMAL state (modular service specification). As a consequence of our automata semantics (see Section 3.2, *State Transition Diagrams (STDs)*), the high-level transition (which is triggered on receiving the action ENABLE) makes the service enter the initial state of the NORMAL MODE. Thus, the service is "ready" for a service call.

Note that we only allow the ENABLE transition to start from the DISABLED state and not from the INTERRUPTED state as the relationships ENABLE and DISABLE logically belong together.

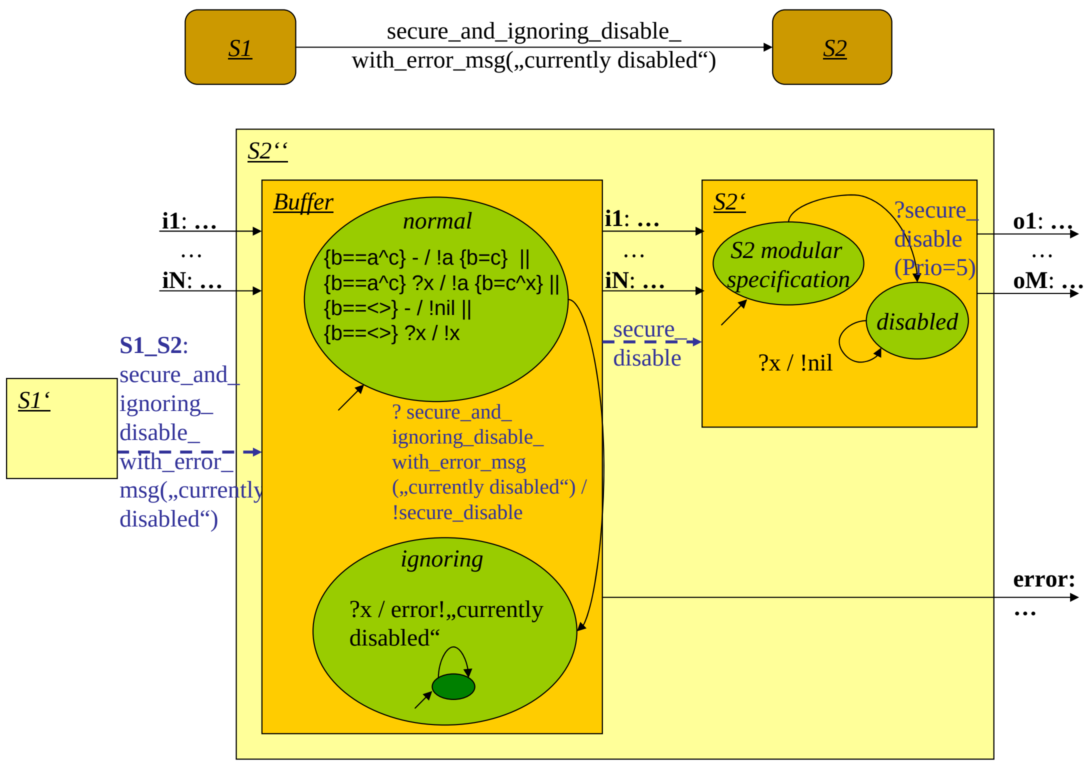

Figure 5.6.: Standard control interface for the realization of the ERROR MESSAGE part (SSD + STD)

5.3.4. Relationship CONTINUE

Figure 5.8 contains the standard control interface for the basic service relationship CONTINUE (as already presented in Section 4.8, *Combination of services on basis of the service relationships*). This time the high-level transition starts from the INTERRUPTED state (and not from the DISABLED state) and leads to the history state of the NORMAL state. As explained in Section 3.2, *State Transition Diagrams (STDs)*, the history state denotes the state in which the service was when the modular service specification was left (due to the reception of an INTERRUPT action).

5.3.5. Combination of standard control interfaces

For the combination of several standard control interfaces (the realization of several basic service relationships) the same holds as already discussed in Section 4.8 (*Combination of services on basis of the service relationships*). For example, conflicts might have to be solved by a conflict solver service.

5.3.6. Discussion on another semantics using buffering channels

In Chapter 3 (*Notational techniques (Overview)*) we introduced the semantics of our notational techniques. We declared channels to be non buffering. Consequently, for

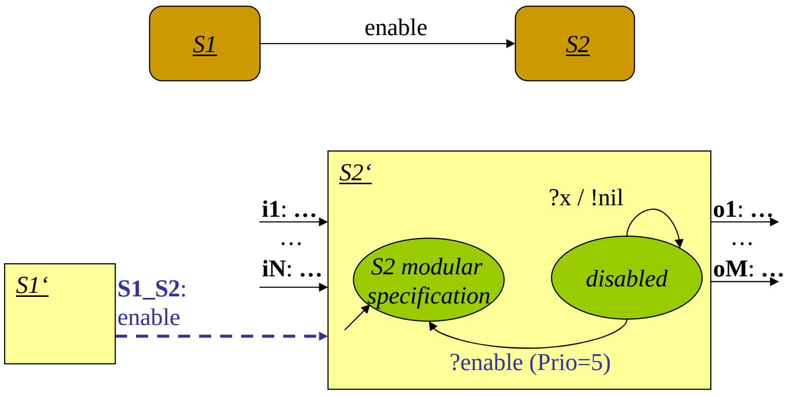

Figure 5.7.: Standard control interface for the basic service relationship ENABLE (SSD + STD)

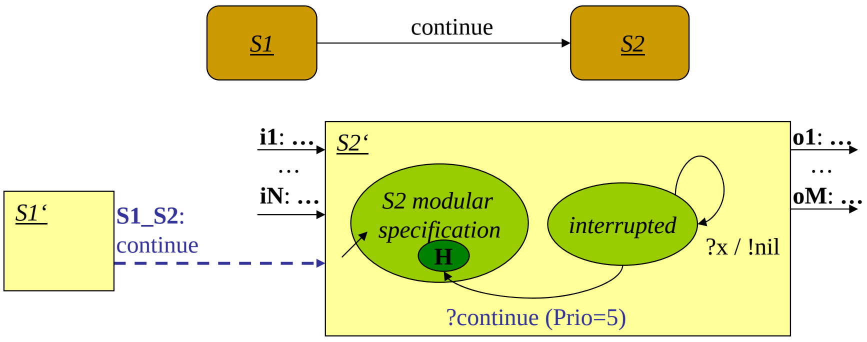

Figure 5.8.: Standard control interface for the basic service relationship CONTINUE (SSD + STD)

those extended basic service relationships which make use of input buffering we explicitly have to model the buffering of the inputs (see Figure 5.4).

Another possibility would be to change the semantics of the notational techniques to buffering channels. In that case, actions would be automatically buffered. It would have to be specified which input is to be handled first, for example the oldest one, or if each buffered input can trigger a transition.

For our methodology we chose non buffering channels as motivated in Section 3.2.1 (*Intuitive description*). If buffering of inputs is the general case for the system under development, it might make sense to make use of buffering channels. Although this is possible, changes to the semantics are non trivial and have effects that would have to be identified and evaluated first.

5.4. Application to the memory cell case study

In the previous section, we introduced the extended set of basic service relationships. In this section we will show their application within our methodology with help of the memory case study.

5.4.1. Starting point, identification of atomic services, and identification of the logical syntactic system interface

The starting point and the first two steps of the methodology (see Sections 4.3, *Starting point*, 4.4, *Identification of atomic services*, and 4.5, *Logical syntactic system interface*) remain the same.

We obtain the atomic services READ and WRITE (performing the read and write requests of the memory), STANDBY ON and STANDBY OFF (with the help of which the system can be switched to the standby mode and back), and the atomic services SYSTEM ON and SYSTEM OFF (for turning the system on and off).

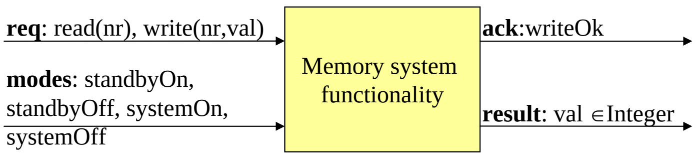
req: read(nr), write(nr,val)
modes: standbyOn, standbyOff, systemOn, systemOff
Memory system functionality
ack:writeOk
result: val ∈Integer

Figure 5.9.: Logical syntactic interface of the memory system (SSD)

The logical system interface of the memory system is depicted in Figure 5.9. Read and write requests are sent on the request channel (abbr. "req"). The parameter "nr" determines the number of the memory cell to be written or read out. The parameter "val" represents the value to be written and the value that is read, respectively.

The first methodological step that has to be adapted to our extended set of basic service relationships is the identification of service relationships.

5.4.2. Identification of service relationships

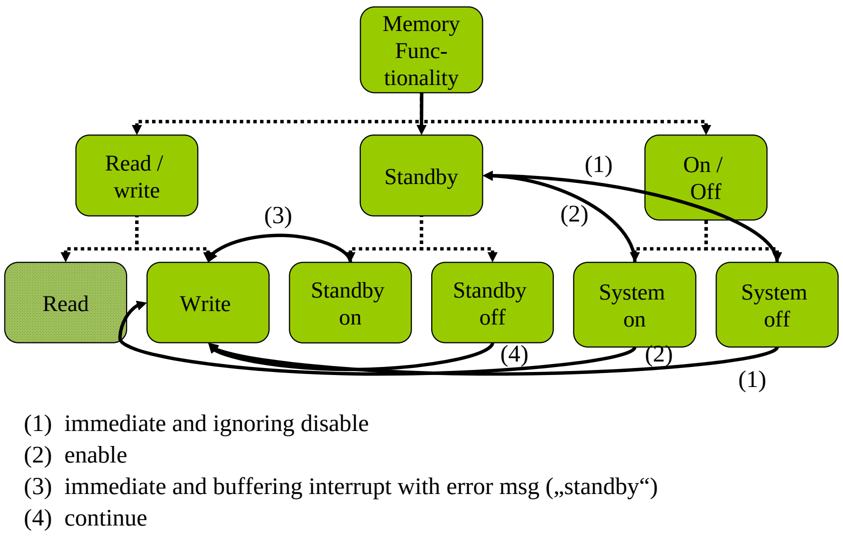

Figure 5.10.: Service graph of the memory system (directed graph)

For our running example we obtain the service graph (including the service hierarchy) shown in Figure 5.10. For reasons of clarity, we did not display the name of the relationships in the service graph, but refer to them by numbers. Also for reasons of simplicity we do not take into account the atomic service READ and the relationships pointing at it.

Requirement 7 demands the IMMEDIATE AND IGNORING DISABLE service relationships (abbreviated by number 1 in the figure). Analogously, Requirement 8 results in the ENABLE service relationship. The basic service relationship IMMEDIATE AND BUFFERING INTERRUPT WITH ERROR MSG ("STANDBY") is a consequence of Requirements 12 and 13. The ENABLE relationship is implicitly given by Requirement 12.

Finally, the additional inner services (READ/WRITE, STANDBY, and ON/OFF) are introduced to further structure the services.

5.4.3. Formal specification of modular services

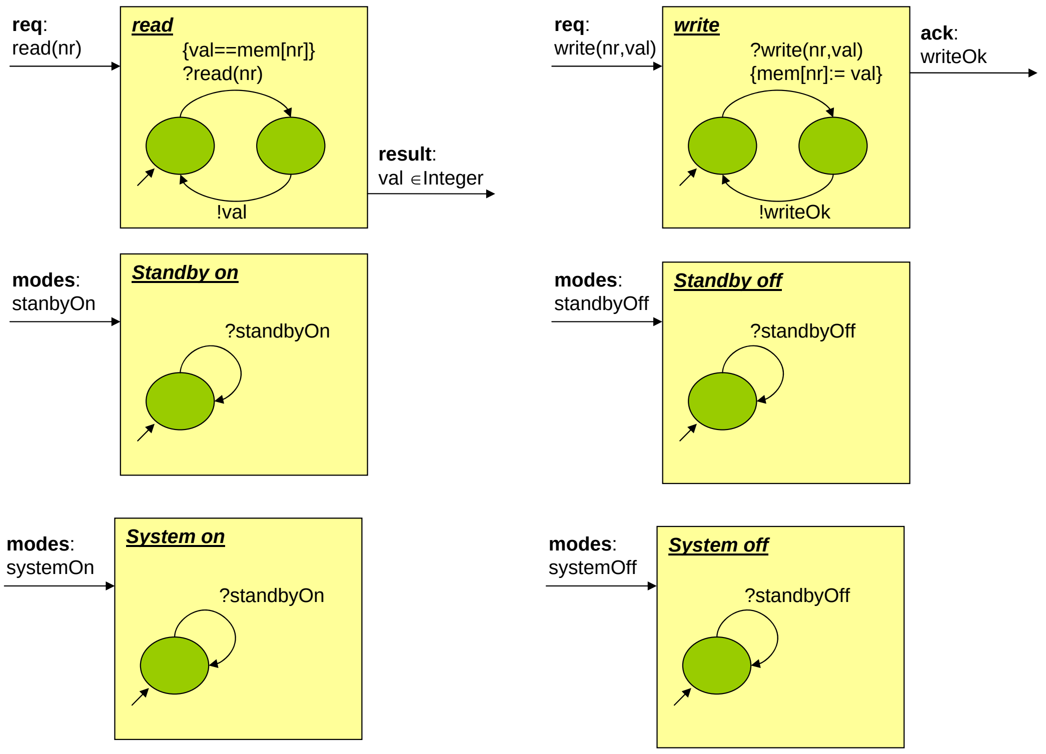

Figure 5.11.: Formal specification of the atomic services of the memory system (SSD + STD)

The formal specification of atomic services is done as introduced in Section 5.4.3 (*Formal specification of modular services*). Figure 5.11 contains the formal models of the modular service specifications. (It is assumed that $1 \leq nr \leq 20$.)

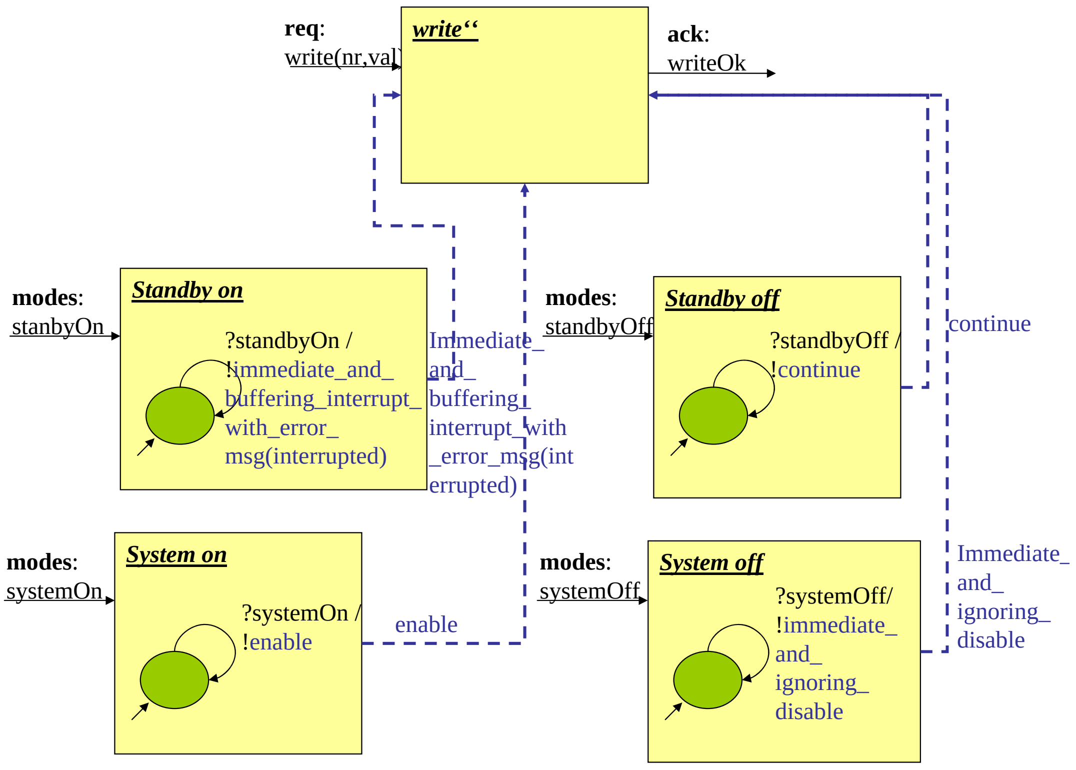

Figure 5.12.: Combination of the memory services - black box view of WRITE service (SSD + STDs)

5.4.4. Combination of services on basis of the extended set of basic service relationships

In Figure 5.12 the combination of the modular service specifications is shown. (Due to reasons of simplicity, the READ service is omitted.) The WRITE service is given from a black box perspective. The white box specification of the WRITE service is (schematically) depicted in Figure 5.13. The STD of the CONFLICT SOLVER service is given in Figure 5.14.

5.5. Discussion on the extended set of basic service relationships

The basic service relationships introduced in Section 4.8 (*Combination of services on basis of the service relationships*) are not powerful enough to model each behavior. Therefore, we introduced an extended set of basic service relationships in this chapter. As we have seen, the extended set is more powerful, however the approach gets considerably more complex. The combination of several standard control interfaces and the conflict solving get more intricate.

Another point is, that some behavior still is not covered by our extended set of basic service relationships. For example, the deleting of the current buffer content on

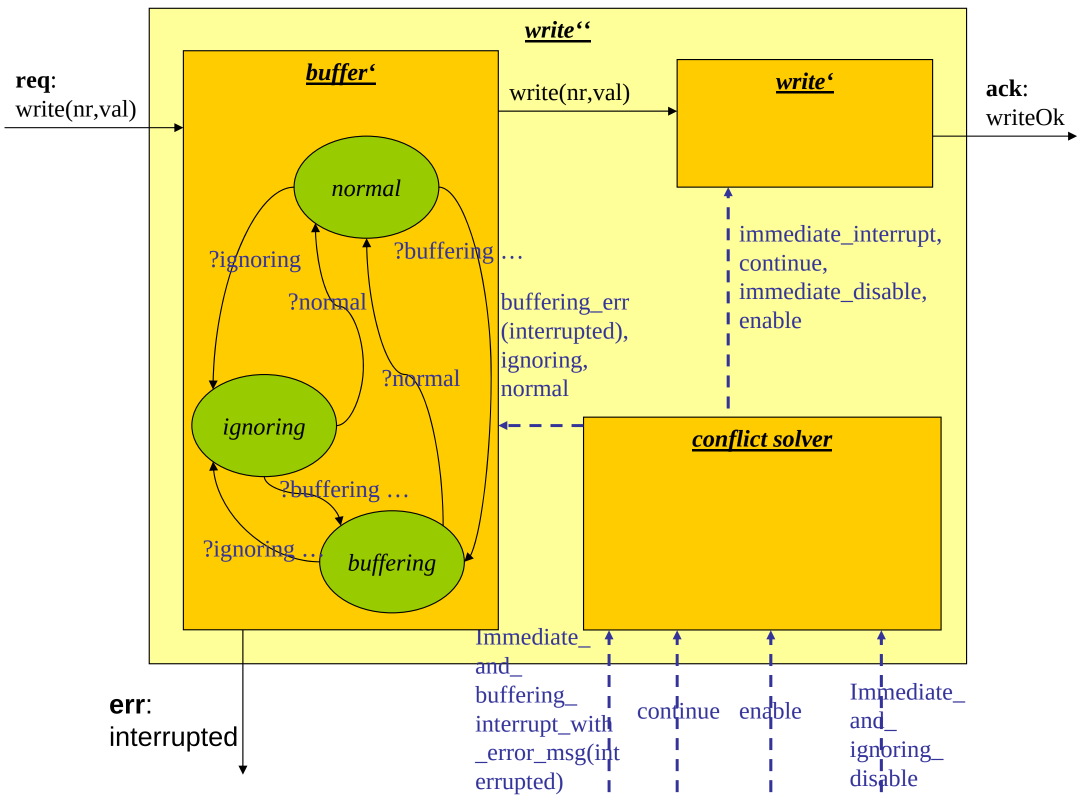
write''
buffer'
write'
conflict solver
req: write(nr,val)
write(nr,val)
ack: writeOk
err: interrupted
normal
ignoring
buffering
?ignoring
?normal
?buffering …
?normal
?buffering …
?ignoring …
buffering_err (interrupted), ignoring, normal
immediate_interrupt, continue, immediate_disable, enable
Immediate_and_buffering_interrupt_with_error_msg(interrupted)
continue
enable
Immediate_and_ignoring_disable

Figure 5.13.: Combination of the memory services - white box view of WRITE service (SSD + STD)

receiving a DISABLE action is not realized but might be wanted in some cases.

Basically speaking, a trade off has to be made. Either the approach is more powerful and complex or less powerful and less complex. Depending on the system this decision has to be made. For behavior that is still not covered by the service relationships, the standard control interfaces have to be adapted manually.

In this chapter, we extended the set of basic service relationships to make our methodology more powerful. We systematically identified and refined the set of basic service relationships, gave a standard control interface for each relationship, and discussed the result. In the following chapter (see Chapter 6, *Related Work*), we compare our methodology to related approaches.

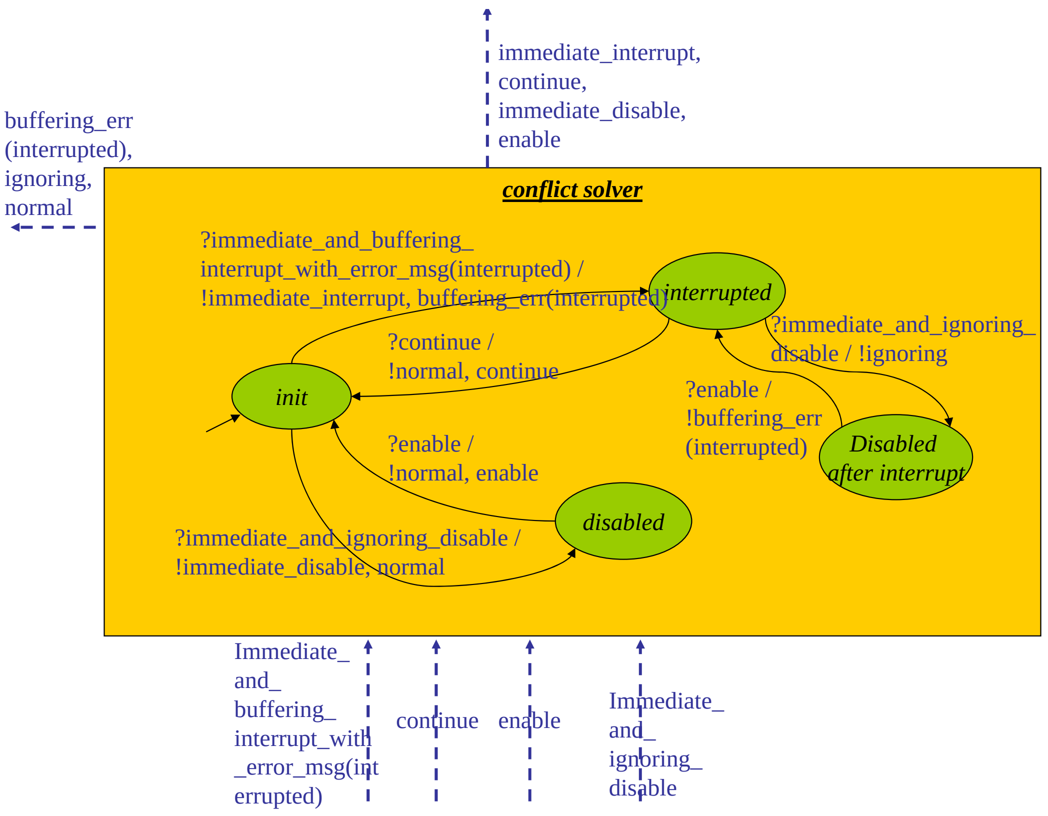

Figure 5.14.: Combination of the memory services (realization of CONFLICT SOLVER, SSD + STD)

Chapter 6

Related Work

In the thesis at hand, we introduced a methodology for modeling the usage behavior of multi-functional systems. In this chapter, we relate our methodology to other approaches.

Our methodology is a *model-based requirements engineering* approach. It stepwise introduces models already to the requirements engineering phase to bridge the gap between the informal requirements engineering phase and the formal design phase. Model-based requirements engineering is a quite young concept. Not many approaches exist so far. We therefore relate our approach to approaches that aim at the seamless transition between the requirements engineering and design phase (see Section 6.1, *Model-based requirements engineering*).

The basic building blocks of our methodology are services. Thus, we presented a *service-oriented development approach*. In the last few years, service-orientation conquered different areas. The underlying concepts (like the notion of service itself) vary more or less from domain to domain. As our methodology is not tailored to a specific domain, but generically applicable, a comparison to service-oriented approaches is difficult. In Section 6.2 (*Service-/Feature-oriented approaches*) we compare our approach to a selected sub set of service-oriented approaches.

In Section 6.3 (*Comparison of the basic system model*) we compare our basic system model (see Figure 4.2).

Contents

6.1. Model-based requirements engineering

Usually, requirements are textually written and thus informally given. Model-based requirements engineering is a relatively young research direction that aims at bridging the gap between informal requirements descriptions and formal models. Although many model-based design approaches exist (model-based design of embedded systems with AutoMoDE [Bauer et al., 2005], model-based design of reactive systems with InServe [TUM, 2006b], model-based development of adaptive and context-sensitive automotive systems with MEwaDis [TUM, 2006c], tool support for model-based development by Rhapsody [Gery et al., 2002], model-based development of embedded systems with IST OMEGA [Graf and Hooman, 2004], model-driven development of enterprise architectures [Kulkarni and Reddy, 2005] - just to mention a few), model-based requirements engineering approaches are quite rare. In this section we relate our approach to two other model-based requirements engineering approaches.

6.1.1. AutoRAID / AutoFOCUS

At this place we relate our approach to the concepts of the model-based requirements engineering tool AutoRAID (AutoFocus Requirements Analysis Integrating Development) [AutoRAID, 2007].

Background information

AutoRAID [AutoRAID, 2007] is a tool for model-based requirements analysis for embedded software systems. Its abbreviation stands for "AutoFocus Requirements Analysis Integrating Development". The tool is integrated with the model-based development tool AutoFOCUS [TUM, 2006d]. The main aim of AutoRAID is to ease the identification, structuring, and consolidation of requirements and to provide a more seamless transition from the informal requirements engineering phase to the formal design phase.

AutoRAID was developed in the summer of 2004 by a team of students and research members of the chair of Software & Systems Engineering, Technische Universität München [TUM, 2006a] and has been continuously enhanced since then.

Approach

As mentioned above, the main aim of AutoRAID is to ease the identification, structuring, and consolidation of requirements and to bridge the gap between the informal requirements engineering phase and the formal design phase.

AutoRAID is integrated with the model-based development tool AutoFOCUS and makes use of AutoFOCUS' formally founded system views. These are: structural view, behavioral (state) view, interaction view, and data view. AutoFOCUS provides the following notational techniques for the specification of the system views, respectively: system structure diagrams (SSDs), state transition diagrams (STDs), extended event traces (EETs), and data type definitions (DTDs). The relationships between these modeling elements (e. g. components, states, transitions, ports, data types) are used for the refinement, structuring, and consolidation of requirements.

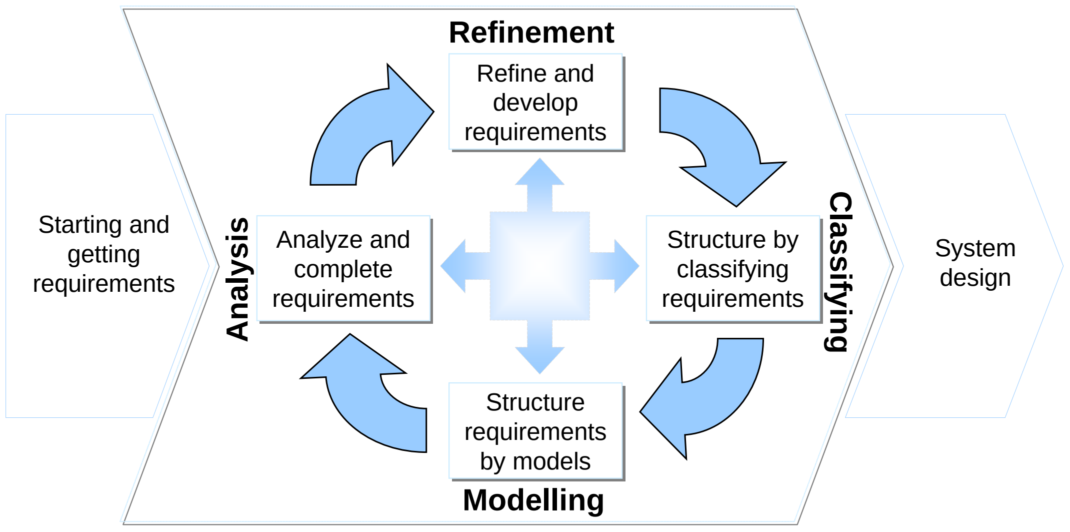
Starting and getting requirements
Refinement
Refine and develop requirements
Analysis
Analyze and complete requirements
Structure by classifying requirements
Classifying
Structure requirements by models
Modelling
System design

Figure 6.1.: Methodological steps of AutoRAID [Geisberger and Schätz, 2007] (graphical overview)

The methodology of AutoRAID is an iterative process which is defined by the following four requirements engineering activities (see Figure 6.1):

- Identification and refinement of requirements: Requirements can be inserted manually or by copying them from an existing file. Further requirements can be derived (refined) from given requirements or from business goals (goal-oriented requirements engineering).
- Classification of requirements: Requirements can be classified according to the different system views (into use cases, architectural constraints, modal constraints, and data type constraints). The refinement and specification is later done specific to the class of requirements.
- Modeling of requirements: The services (functions) of the system are refined and modeled iteratively. The steps within use cases are "observed" and captured in the elements of the design model (interaction, mode switch, data operation). Furthermore, model elements are "motivated" by constraints (e. g. introduction of components or states).
- Analysis of requirements: The core techniques of AutoRAID for the analysis of requirements are the constructive support of refinement, tracing mechanisms, and the detection of missing requirements (due to the association to model elements).

Commonalities and Differences

Both AutoRAID and the approach of this thesis aim at a seamless transition from textual requirements to formal models. AutoRAID takes into account all kinds of requirements (i. e. business requirements, functional requirements, system requirements); the methodology presented above only takes care about the modeling of functional requirements.

The main idea behind AutoRAID is to relate requirements with elements of the *design model*. Thus the focus is on vertical tracing (relating artifacts of different phases/stages of the development process). In the approach of this thesis, a *requirements model* is created. The focus therefore is on horizontal tracing (relating artifacts of the same phase/stage of the development process). Consequently, AutoRAID deals with models representing different system views (structure, internal behavior, interaction, and data) whereas our methodology primarily makes use of behavioral models.

One of the main ideas behind the presented approach is to make dependencies between functionalities (services) explicit. In AutoRAID two different kinds of relationships exist. First, so-called "motivation" and "association" links relate functional requirements to elements of the design model. However, the "motivation" and "association" links of AutoRAID are coarse and only express the existence of a relationship. Second, functional requirements can be (assigned to business requirements and) structured hierarchically. In our methodology, we introduced differentiated basic service relationships and horizontal service relationships. Furthermore, we made use of the (restricted) sub service relationship to structure services hierarchically.

The stepwise transformation of textually given steps of sequence diagrams in AutoRAID, resembles our step of the formalization of modular services. In our methodology, single services are modeled by automata.

The approaches differ in the main idea on which we based our approach: In our methodology we show how the modularly specified behavior (given by functional requirements) has to be adapted because of other functional requirements (services). This is not looked at in the approach behind AutoRAID.

6.1.2. Unified Modeling Language 2.0 (UML 2.0)

It has been proven that modeling is an essential part of medium- to large-scale software projects. Many modeling languages have been developed during the last years to support the development process. The most famous one is probably the Unified Modeling Language (UML) [OMG, 2003].

Background information

The Unified Modeling Language (UML) was developed by the Object Management Group [OMG, 2007a]. It has become the de facto standard for modeling object-oriented software systems. The first version of the UML appeared in 1990 as a reaction to numerous suggestions for modeling languages. A revised version of the UML (UML 2.0) was released in 2005.

Approach

The UML informally defines identifiers for most notions which are important for the modeling process and also identifies relationships between them. There exist 13

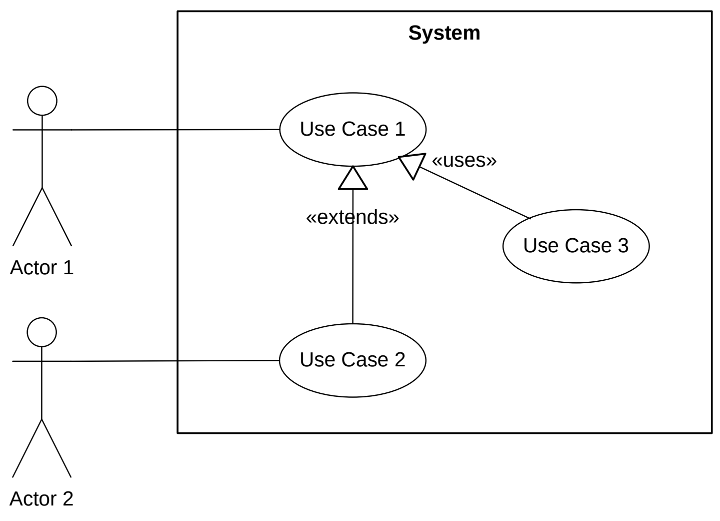

Figure 6.2.: Example of a UML Use Case Diagram (UML Use Case Diagram)

standard diagram types for the graphical representation of parts of the UML model. Furthermore the UML suggests a format for the interchange of models and diagrams between different tools.

The standard diagram types are divided into *structure diagrams, behavior diagrams,* and *interaction diagrams* which are used to model different aspects, respectively. The diagram type which is most similar to our approach are UML Use Case Diagrams. Figure 6.2 contains an example for a UML Use Case Diagram. Actors indicate human users or surrounding systems (with which the system communicates). Ovals represent use cases (functionalities). Use cases are connected (by lines) with those actors that affect the use case or that are affected by the use case. UML Use Case Diagrams provide two different kinds of relationships between use cases: $<< extends >>$ and $<< uses >>$. The extend relationship helps to reduce the complexity of use cases. The extend relationship visualizes the effect that an instance of an extended use case may include (under certain circumstances) the flow of events specified by the extending use case. The use relationship is used to represent commonalities in use cases. A use case is used by another use case if it is included in its flow of events.

Commonalities and Differences

The UML 2.0 informally defines notions and relationships between notions. It is relatively easy to learn and supports several views onto software systems. However, the simplicity of the Unified Modeling Language has its costs: Its concepts are not formally founded. Furthermore, the UML itself does not present methodological support.[1] We base our work on a formally founded system model. Furthermore, we provide the semantics of our service relationships by giving schemes (automata) how to realize them.

We do not take into consideration actors. If necessary, actors can be easily added

[1]Although numerous methodologies were developed around the UML.

to our diagram types (service hierarchy/graph and SSDs). In our methodology we also separate modular functionalities from relationships between them. However, we distinguish between several relationships which are observable at the system boundaries and - as mentioned above - give their semantics.

6.1.3. Systems modeling langague (SysML)

We now relate our methodology to the work of the SysML project [OMG, 2007b].

Background information

In 2001 the Object Management Group (OMG) together with the International Council on Systems Engineering (INCOSE) founded the Systems Engineering Domain Special Interest Group (SE DSIG). The aim of SE DSIG was to develop a standardised enhancement of the UML to make it a modeling language for the specification, design, and verification of complex systems. In 2003 a request for proposal was published containing the requirements for such a modeling language. The SysML workgroup formed in order to answer this proposal. Members of the SysML group for example are: IBM, Telelogic, I-Logix, Motorola, NASA, and EADS.

SysML was accepted as standard by the OMG in April 2006. The current SysML specification can be obtained at [OMG, 2007b].

Approach

SysML is an enhancement of UML. It is a general-purpose graphical modeling language suitable for specifying, analyzing, designing, and verifying complex systems. The systems may consist of hardware, software, information, processes, personnel, and facilities [OMG, 2006]. SysML provides graphical notational techniques for modeling requirements, the system behavior, and the system structure. It is specified by a combination of UML modeling techniques and "precise natural language" [OMG, 2006].

SysML is based on UML 2.0. Its most important enhancements (above others) are [oose Innovative Informatik, 2007]: structure diagrams, requirements diagram (modeling of functional and non-functional requirements), parametric diagram (modeling of parametric relationships between model elements). For the comparison of SysML and our approach, only the requirements diagram is of interest.

The requirements diagram is used to represent text-based requirements. It relates them to model elements of other diagrams. Requirements can be depicted either in graphical, tabular, or tree structure format. Additionally, a requirement can appear on other diagrams in order to show its relationship to elements of design models. SysML introduces several requirements relationships including relationship for relating requirements hierarchically (composite requirements, subrequirements), deriving further requirements, satisfying requirements (a model element can satisfy a requirement), verifying requirements (a test case can verify a requirement), and refining requirements.

Commonalities and Differences

Analogously to AutoRAID, SysML relates requirements with elements of the design models. For example, the relationships $<< satisfy >>$ and $<< verify >>$ are used to relate textual requirements to design elements and to test cases, respectively (horizontal tracing). In contrast, in our approach we make use of behavioral models within the requirements engineering phase.

Both approaches make use of hierarchies (and graphical representations of hierarchies) in order to structure functionalities (functional requirements and services, respectively).

However, again, the main idea underlying the methodology of this thesis (namely that the modularly specified service behavior has to be adapted according to relationships) is not looked at.

6.2. Service-/Feature-oriented approaches

In this section we relate our methodology to other service-oriented approaches. We hereby concentrate on a selected set of approaches which seem to be similar to our approach. A comparison of different service-oriented approaches can be found in [Meisinger and Rittmann, 2008].

6.2.1. Services in the telecommunication domain - Distributed Feature Composition (DFC)

Many roots of service-orientation lie in the area of telecommunication. In this section we compare our approach with one prominent approach of this domain, the one of Pamela Zave and Michael Jackson [Zave and Jackson, 2000, Zave, 2003, Zave, 2001].

Background information

The approach presented in this subsection is the creation of Michael Jackson and Pamela Zave. Michael Jackson (currently) is a visiting research professor at the Department of Computing, Open University, and a visiting professor of the School of Computing Science, University of Newcastle. Pamela Zave (currently) works at AT&T Labs Inc.-Research as Technology Consultant.

Approach

Distributed Feature Composition (DFC) is a virtual architecture. It aims at the specification and implementation of telecommunication system. It was developed as a response to the feature-interaction problem (see below) in telecommunication systems.

The approach assumes that a base of functionality already exists. This might be an existing implementation of a POTS (Plain Old Telephony System) or any other sys-

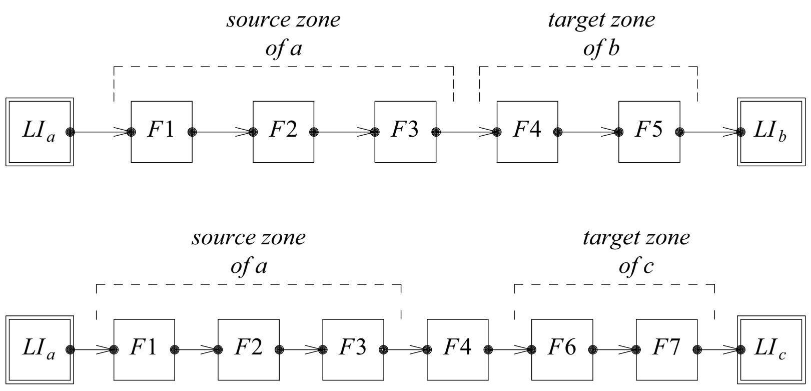

Figure 6.3.: Linear usage within Distributed Feature Composition [Zave, 2001]

tem. The main goal of the feature-oriented approach is how to add, remove, modify and combine pieces of functionality later in the life cycle of such systems.

A *feature* is an "optional or incremental unit of functionality" [Zave, 2001] and resembles our notion of service. The feature specification contains an action, enabling condition and priority. The action is performed if the enabling condition is true and the priority is the highest. A feature-oriented description is a "description of a software system organized by features, consisting of a base description and feature modules, each of which describes a separate feature" [Zave, 2001].

Zave and Jackson define the term *feature interaction* as "some way in which a feature or features modify or influence another feature in describing the system's behavior set" [Zave, 2001]. They explicitly distinguish between wanted and unwanted feature interaction. Unwanted results of feature interaction are incompleteness, inconsistency, non-determinism, and unimplementability. Desired feature interaction "can be" achieved without changing any feature modules, rather by simply adjusting the precedence relation; i. e. the order in which feature can occur in a route (see later).

The following methodology accompanies DFC:

1. Describe new features as if they were independent (manually).
2. Understand all potential interactions (help of automated analysis necessary).
3. Classify interactions as bad or good (manually).
4. Adjust feature descriptions (priorities) so that the result contains no more bad interactions.

DFC is a virtual architecture with the basic goal to support usages. A usage is a dynamic assembly of features, line/device interfaces and internal calls to satisfy a system's requirements to connect two end points on behalf of a user with a given set of requested features (see Figure 6.3). A router component is responsible for establishing a connection between the end points across the source and target zones, activating a certain number of features along the call. All features have a certain activation protocol and can be invoked independently from each other. Features are connected to each other and to end points by internal calls. To avoid or control feature interaction, all features have a priority that determines the order in which they can be applied by the router. The basic idea behind DFC features is that they are autonomous, modular units that can be applied in sequence by adhering to a standard black-box feature interface. In this way, the DFC architecture follows the "Pipes-and-Filters" architectural style described in general in [Buschmann et al., 1996].

The precedence relation defines the order in which features occur in a route. It is the only place of the algorithm where feature relations are captured. The goal is to place features with a higher priority later in the route, close to the target zone. The proponents of the methodology claim that their concept of features and feature precedence provides a useful degree of behavioral modularity and that it is possible to manage desired and unwanted feature interactions by simply adjusting the precedence relation without modifying any of the feature modules themselves [Zave, 2001].

Commonalities and Differences

When comparing the DFC with the methodology of this thesis, we see that the idea is the same: Services (Features) are specified modularly in order to reduce complexity and enhance reuse. Services (Features) may influence each other; consequently we face the problem of feature interaction.[2] In the approach by Zave/Jackson the problem of feature interaction is solved by introducing a partial ordering on features according to their priorities. In our methodology we adapt the behavior in order to handle feature interaction.

Zave/Jackson explicitly distinguish between wanted and unwanted feature interaction. In our methodology, we only take care of wanted feature interaction as specified in the functional requirements. As mentioned in Section 7.3 (*Outlook*), a calculus on service relationships would help in identifying implicitly given service relationships. These implicitly given relationships make up unwanted feature interaction. Given such a calculus on service relationships, our methodology could be enhanced to also deal with unwanted feature interaction.

In DFC features are added to a base specification. In our approach, we combine services (features) without a base specification. Furthermore, DFC does not introduce differentiated service relationships. In contrast, the (vertical and horizontal) service relationships are the basis for the combination process.

6.2.2. FODA, FORM, FOPLE

Service-/Feature-oriented software development is also common in the field of product line approaches. In this subsection we relate our approach to the first feature-oriented product line approach - Feature Oriented Domain Analysis (FODA) [Kang et al., 1990] and two of its enhancements (FORM and FOPLE).[3]

[2]In fact, we adopted the definition of the term feature interaction [Zave, 2001] for our approach.

[3]A good summary of the FODA methodology can be found in [Adersberger, 2006] (in German).

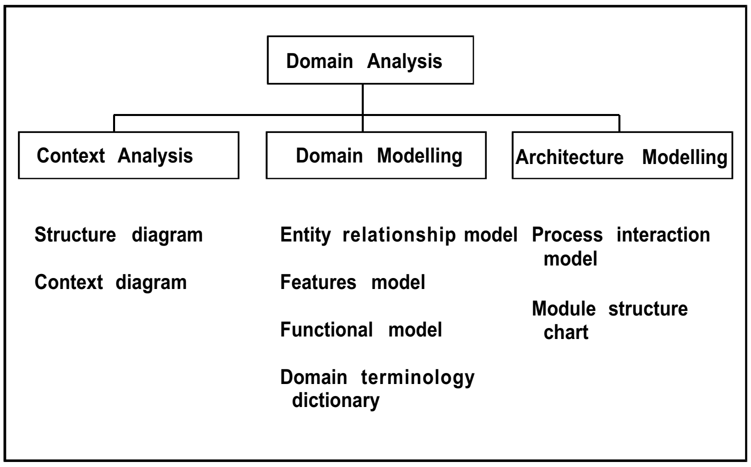

Figure 6.4.: Phases and products of domain analysis

Background information

FODA was developed at the Software Engineering Institute (SEI), Carneggie Mellon University, in 1990 under guidance of Kyo-Chul Kang. The aim of FODA was to develop a methodology for the identification and specification of commonalities and differences within software systems (the term "product line" was not used at that point of time).

There existed several methodologies for the domain analysis of software systems already in the 80ies. However, these methodologies were not applied in practice but were only a matter of research. FODA managed to be applied in practice by publishing its concepts by a feasibility study [Kang et al., 1990] instead of by publishing a theoretical concept paper.

FODA first was an independent research domain within the SEI. In 1994 FODA was embedded in the MBSE initiative (Model-Based Software Engineering) [Withey, 1994]. In the meantime, FODA and MBSE are declared to be "legacy". FODA now is one of six methods suggested in the *Software Engineering Practice Area: Domain Analysis* in the *Pruduct Line Practice (PLP)* Framework [CMU, 2007].

Both the Feature-Oriented Reuse Method (FORM) [Kang et al., 1998] and Feature Oriented Product Line Engineering (FOPLE) [Kang et al., 2002b] are enhancements of the FODA methodology.

Approach

Figure 6.4 shows the three phases of the FODA method and lists the product of each. The domain analysis starts with the **context analysis** the aim of which is to make a scoping with regard to the domain. Two products are the result of the context analysis: The context diagram describes the data flow between the system under specification and surrounding systems. The structure diagram positions the domain with regard to other domains.

The aim of the **domain modeling** is to model the problems of the domain and document the domain knowledge. The first part of the domain modeling phase is the feature analysis. It is the most famous part of the FODA methodology. The aim of the features model is to provide an overview of the capabilities and features of the system under development. The features model is the main instrument for the communication with the end user of the system. During the information analysis, the domain knowledge is captured by entities and their relationships (entity relationship model). During the functional analysis, commonalities and differences of applications within a domain are identified. The so-called functional model is created. In this phase, the largest part of the domain glossary is also usually created (domain terminology dictionary).

In the last phase of the FODA methodology, the **architecture modeling**, the solution space is described. For example, reference architectures for the domain are described.[4] The architecture modeling phase is covered shortly in [Kang et al., 1990]. It is based on the DARTS method as presented in [Gomaa, 1984]. The main idea is to describe the system architecture by two layers: the process interaction model and the module structure chart. The former describes the domain processes and its relationships. The latter contains the decomposition of the entities (of the entity relationship model) and the functions (of the functional analysis) into modules.

FORM is an enhancement of FODA to elaborate the architecture modeling phase. Furthermore, the features model was enriched for possibilities to more formally specify the system features. FOPLE is an interpretation of FORM with regard to product line development. The main enhancement is the introduction of a marketing and product plan (MPP). The MMP substitutes the context analysis by a market analysis.

Commonalities and Differences

FODA describes a methodology targeting at the requirements engineering and design phase. Our methodology only deals with the requirements engineering phase. Furthermore, FODA is a product line oriented approach whereas we aim at specifying one system.

FODA adopts the definition of [Pickett, 2000] and defines a feature as a "prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems". In our approach, a service is a functional, user-visible characteristic of a software system. Therefore, our services can be seen as a subset of the FODA features - namely the FODA features that describe functionality.

In FODA, the scoping is done by means of the context analysis. The determination of the syntactic interface (step 2 of our methodology) can be compared with the context diagram of FODA.

The features model of FODA is an AND/OR tree and resembles our service hierarchy. FODA also distinguished between vertical and horizontal relationships between the features model.There are three different kinds of vertical relationships: $<< consistof >>$ (a feature consists of other features that have to exist in the system), $<< optional >>$ (the existence of a feature is optional), and $<< alternative >>$

[4]An example is the QUASAR reference architecture for business information systems [Siedersleben, 2004].

(only one feature can be chosen for a concrete system). Additionally, the following horizontal relationships exist: $<< mutuallyexclusivewith >>$ and $<< requires >>$. The FODA relationships are motivated by product line concepts. They mainly describe dependencies between features that determine how concrete products can be assembled. In FODA, the features are structured by a tree. In the approach presented in this thesis, a service might also have several parents.

As in FODA, a feature is also a user-visible characteristic, the (functional features of the) features model also describes a black-box view onto the system. However, in FODA no formal model within the requirements engineering phase is created, i. e. neither the relationships are formally founded nor the service specifications are given formally. Furthermore, the relationships are a means to assemble a product rather than to combine feature in order to obtain a formal model of the overall behavior.

6.2.3. MEwaDis

In this subsection, our methodology is compared to another model-based development process - MEwaDis.

Background information

MEwaDis (German: Modellbasierte Entwicklung adaptiver Dienste, English: model-based development of adaptive services) [TUM, 2006c] was funded by the Hightech-Offensive Zukunft Bayern, BMW Car IT, ESG and Peak-Systems. The primary aim of MEwaDis was the development of techniques and methods for the analysis, modeling, and validation for the development of reliable, adaptive, context-sensitive services.

Project partners were BMW Car IT GmbH, BMW Forschung und Technik GmbH, and the chair or Software & Systems Engineering, TUM.

Approach

In [Deubler et al., 2004b, Deubler et al., 2004c, Deubler et al., 2005, Deubler et al., 2004a] the service-based development process as elaborated in MEwaDis is presented. The process is made up of the phases shown in Figure 6.5. In the **Inception Phase**, the project mission and requirements are elaborated. Afterward, the sequence of actions are modeled in the **Service Identification Phase**. This is done on a high level of abstraction. Activity diagrams are used to describe the run of service functions. In the subsequent **Use Case Modeling Phase**, the services of the Identification Phase are elaborated. To that end, the flow of events of each service function is specified. Furthermore, inputs and outputs, preconditions, security requirements, and service requirements are captured. The result of this phase is a use case model with textual descriptions of the use cases, sequence diagrams for a service function, a textual description of the security requirements, and a logical service architecture.

In the **Service Modeling Phase**, the behavior of a service is formally specified by

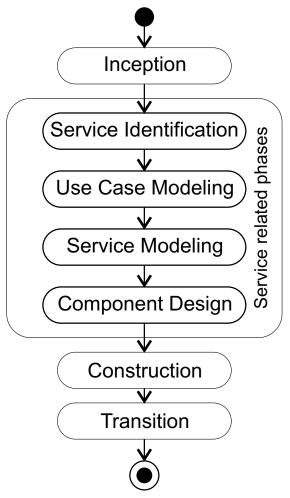

Figure 6.5.: Phases of the service-oriented development process of the MEwaDis process [Deubler et al., 2004c]

state transition diagrams. Moreover, the security requirements are concretisized. The logical service architecture is mapped to a system architecture in the **Component Design Phase**. As result of this phase, we obtain a behavioral model of the system components with assigned services.

The last two phases - **Construction** and **Transition** - can be carried out conventionally as know from other development processes and are not specific to services.

Commonalities and Differences

For a comparison of the MEwaDis approach and our methodology, only the Service Identification, Use Case Modeling, and Service Modeling Phases are of interest.

The service specifications of the MEwaDis approach are comprised of the syntactic interface, the semantic interface, and quality of service attributes. Quality of service attributes are not looked at in our methodology.

Furthermore, the service specification of MEwaDis explicitly contains relationships to other services. In the approach presented in this thesis, the service relationships are separated from modular service specifications. Additionally, there are no detailed service relationships or defined semantics of the relationships in the MEwaDis approach. The service specifications only refer to "Involved Services".[5]

Moreover, MEwaDis does not stick to a black box view onto the system behavior, as the internal communication (e. g. call relationships) between services is also specified.

The Service Identification Phase of the MEwaDis methodology could be adapted for our approach to identify services out of given requirements.

[5]It is suggested to integrate service relationships into the CASE Tool AutoFOCUS, (see Section 6.1.1, *AutoRAID / AutoFOCUS*). It should be possible to annotate service relationships to services so that this information can be used in later phases of the development process, i. e. for model transformations.

Both approaches make use of System Structure Diagrams (SSDs) and State Transition Diagrams (STDs) to model services.

6.2.4. Functional architecture modeling

In this subsection we describe the approach of Bernhard Schätz et al. as described in [Schätz, 2005, Schätz and Salzmann, 2003] and [Kof et al., 2004].

Background information

The focus of this work is the explicit modeling of functional architectures. The type of the target systems are distributed reactive systems. During the last years, several publications on this topic have occurred.

Approach

The main idea behind the presented approach is the following: Different functions of a system are first specified separately (modularly). Then, these modular specifications are integrated into a functional architecture. The integration is done on basis of well-defined, formally founded operations to ensure compatibility and consistency of the functions. Services encapsulate pieces of functionality. A service is comprised of its interfaces (*input ports* and *output ports*), its variables (data space), its configurations, and its transitions. Services communicate with each other over typed and directed channels connecting input ports to output ports.

The behavior of a service is described by *configurations* and *transitions*. A configuration is a state of a system describing which (sub) services are active in a configuration. A transition describes how a services changes from one transition to another. So-called *connectors* describe the entry and exit points of a transition, i. e. define how a configuration is activated and terminated.

The behavior of a service is described by means of the following notations: a *state* is an assignment of input and output ports as well as variables to their current values. A *step* is a pair of states describing the change between the two states. An *observation* is either a finite or an infinite sequence of states with a given starting location and - in case of a finite sequence - an ending location. The *behavior* of a service is the set of observations which are performed by the service.

By using *disjunctive combination*, services are combined to form alternative configurations. The disjunctive combination of two services results in a service making use of the sum of input and output ports and the sum of variables. It exhibits the combined behavior of each service. The *conjunctive combination* of two service exhibits the joint behavior of each service.

Commonalities and Differences

Compared to our approach, we obtain the following commonalities: The approach of Schätz et al. also uses the concept of services to encapsulate pieces of *partial* functionality. Services are used to describe the functional architecture of reactive systems. Additionally, the methodology puts the integration of functionality at an early stage in the development process. Both approaches use the AutoFOCUS [Huber et al., 1997] computational model as basis.

In our methodology we combine the services based on the service relationships between them. Thus we emphasize on modeling the various dependencies between services. We also focus on how to adapt modular service specifications in order to handle feature (service) interaction.

6.2.5. Formal foundation of service-orientation (FOCUS theory and JANUS approach) and the VEIA approach

The JANUS approach [Broy, 2005] (based on the FOCUS theory [Broy and Stolen, 2001]) is the formal foundation of our concepts. In the Appendix A (*Embedding into a theoretical framework*) we embed our approach into this formal framework. Furthermore, we compare our approach to the predicate-based, service-oriented approach by the VEIA project which is also based on the concepts of FOCUS and JANUS.

Both works are just mentioned at this place for the sake of completeness.

6.3. Comparison of the basic system model

In Section 4.1.3 (*Specialties with embedded systems*) we introduced our basic system model. In this section we compare it with the Four Variable Model by Parnas and Madey.

Background information

The Four Variable Model (FVM) was introduced by Parnas and Madey [Parnas and Madey, 1995] to organize software documentation. The authors both give a formal foundation for the FVM and a list of documents which are needed to completely document the system-to-be. Other approaches enhance the FVM by providing software tools or requirements development guidelines, e. g. the Software Cost Reduction method (SCR, [Heitmeyer, 2002]) and the Core Method for Real-Time Requirements (CoRE, [Faulk et al., 1992]).

Approach

The basic idea is to document a system by describing it in a mathematical way. The system behavior is specified by functions relating exactly one element of the domain to one element of the range.

To that end, four variables are defined:

- **Input** variables represent the hardware input provided by sensors, switches, etc. For example, an input from a speed sensor may be a 16-bit unsigned quantity.
- **Monitored** variables capture environmental quantities from inputs. They are described in terms of the domain vocabulary. For example, the current speed of the car may be expressed by a variable "velocity".
- **Controlled** variables represent environmental quantities from outputs. Analogously to monitored variables, they are expressed in terms of the domain vocabulary. For example the angles of the seat position (of the driver's seat) may be described by a controlled variable.
- **Output** variables provide values that the hardware understands. For example, a special floating-point value sent to an actuator may set the angles of the seat to the required position.

Furthermore, four functions are specified on these variables:

- **IN** maps monitored variables to input variables and thus describes the behavior of the input devices (e. g. sensors or switches).
- **REQ** maps controlled variables to monitored variables and thus describes the behavior which the system is supposed to expose.
- **OUT** maps output variables to controlled variables and thus describes the behavior of the output devices (e. g. actuators).
- **NAT** maps monitored to controlled variables and thus defines the environmental context of the system.

Commonalities and Differences

The Four Variable Model describes the system by mathematical functions mapping elements of the domain to the range. In our methodology, these mappings are described by automata which process inputs and produce outputs.

In case our methodology is used to model the overall (possibly further decomposed) system, we abstract from the mappings IN and OUT of the FVM. We then use the monitored and controlled variables for the definition of input actions and output actions, respectively. In the subsequent system design the system-to-be would be divided into hardware (including input and output devices). The model would then have to be refined accordingly.

In case our approach is used for the specification of an embedded system (being surrounded by other system entities within the overall system) the logical input and output actions represent the input and output variables of the FVM. The design specification of the overall system would have to take care of the mapping of monitored to input variables and controlled to output variables.

In this chapter, we compared our work to related approaches. In the following chapter (Chapter 7, *Summary, evaluation, and outlook*) we summarize our methodology, give an evaluation of it, and list future research topics.

Chapter 7

Summary, evaluation, and outlook

In this chapter, we summarize our approach again briefly. Furthermore, we evaluate the approach by listing a set of its advantages and disadvantages. Finally, we give an outlook on future research topics.

Contents

7.1. Summary

In this thesis, we introduced a methodology for the modeling of usage behavior of multi-functional systems. Figure 7.1 summarizes the approach graphically.

The *starting point* of our approach are functional requirements which are given textually, i. e. in an informal form.

The first three steps define the *informal phase*:

In the first step, the *atomic services* which are the "smallest" user-visible services of the system are identified. They are specified textually by means of a table. Furthermore, persistent data needed by the atomic services is informally specified in a table repository.

Afterward, the *logical syntactic system interface* is determined. It is comprised of a set of *logical input* and *logical output channels* and their *logical input actions* and *logical output actions*, respectively. The channels and actions are specified in table form and by means of an SSD.

In the subsequent step, the service relationships are identified. We distinguish between *vertical service relationships* and *horizontal service relationships*. Vertical relationships hierarchically decompose the overall system functionality into sub services. We specify the vertical relationships graphically by means of a *service hierarchy*. Additionally, horizontal relationships are identified. We hereby identify a set of *basic*

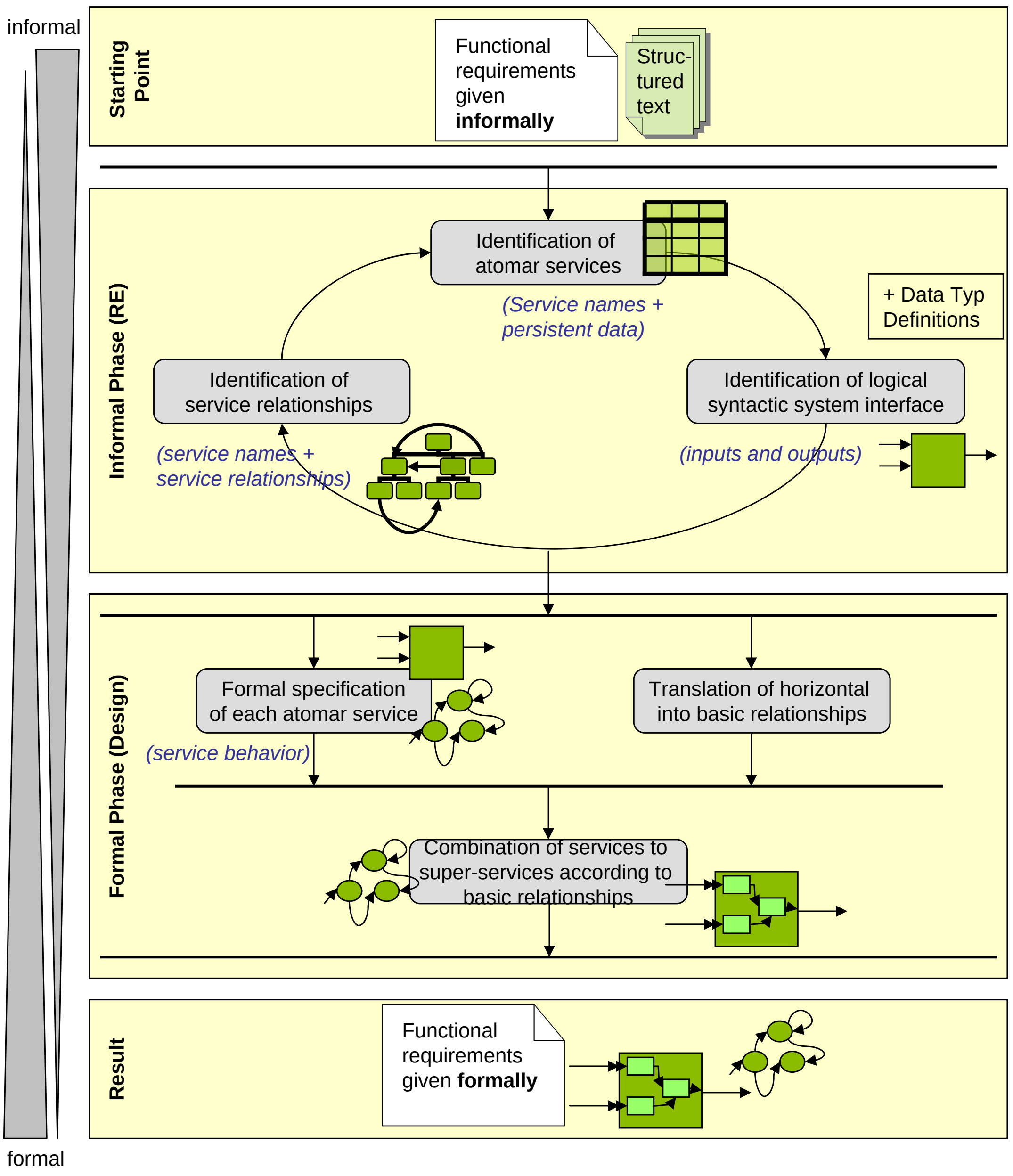
informal
formal
Starting Point
Functional requirements given **informally**
Struc-tured text
Informal Phase (RE)
Identification of atomar services
(Service names + persistent data)
+ Data Typ Definitions
Identification of service relationships
(service names + service relationships)
Identification of logical syntactic system interface
(inputs and outputs)
Formal Phase (Design)
Formal specification of each atomar service
(service behavior)
Translation of horizontal into basic relationships
Combination of services to super-services according to basic relationships
Result
Functional requirements given **formally**

Figure 7.1.: Steps of the methodology (activity diagram)

service relationships and *(complex) horizontal relationships* which can be translated into basic relationships. It might be necessary to further specify the horizontal relationships by means of parameter. Data dependencies between services that operate on the same persistent data are also captured. The sum of all service relationships is graphically represented in the *service graph*.

By performing the next steps of our approach (the *formal phase*), we stepwise become more formal until a formal model of the system behavior is obtained:

First, the atomic services (which are the leaves of the service graph) are formally specified. Their *syntactic interface* is specified by an SSD, their *semantic interface* is specified by an STD, respectively. The persistent data is specified formally, too, e. g. data types are defined (if needed) and assigned.

Simultaneously, the complex horizontal service relationships are translated into basic service relationships.

Finally, the modular formal service specifications are combined. To that end, we make use of a *bottom-up approach* concerning the service graph. Services which are responsible for realizing the horizontal relationships are introduced. Depending on which basic relationships point to a service, we stepwise adapt its behavior by implementing standard control interfaces. As some relationships might depend on the execution status of the services, we also have to adapt the modular service specifications to provide the respective information. Furthermore, conflict solving services are introduced to resolve conflicts. Additionally, conflicts because of services which concurrently write the same persistent data have to be eliminated.

The result of our approach is a formal model of the overall system functionality from a black box perspective.

In our methodology the system functionality is comprised of "smaller" services. Thus, all services except the atomic services have an inner structure. As already explained in 4.2 (*Overview of methodological steps*) this might lead to the assumption that our approach is not a black box approach. However, as we only model the observable usage behavior of a system we consider our methodology to be a black box approach.

In the appendix (Appendix A, *Embedding into a theoretical framework*)) we embed our approach into a formal framework: the FOCUS/JANUS theory [Broy and Stolen, 2001, Broy, 2005] and compare it to a predicate-based service-oriented approach.

7.2. Evaluation

In this section, we evaluate our approach by listing its advantages and disadvantages. Furthermore, we describe our experience with the application of our methodology.

7.2.1. Advantages of the approach

In Section 1.3 (*Contributions of this thesis*) we already mentioned some advantages of our approach. We again summarize them shortly:

We defined our process with the help of methodological steps. Therefore, the user is given a *systematic procedure*. The output of each steps is well-defined. These outputs can be used to control the intermediate results of the process.

The process allows the *enhanced reuse* as the approach primarily is based on the concept of modularity. Both modular service specifications and a catalog of complex horizontal service specifications (defining how these complex relationships can be translated into basic relationships) can be reused. Other artifacts, like the specification of actions or channels, of course, can be reused, too.

Our approach makes relationships between services explicit. Therefore, the complex interplay between services can be understood (the *"big picture"* is sketched out). The dependencies between functionalities and their effects on the core behavior of services (modular service specifications) is modeled. Furthermore the dependencies between service relationships is modeled. For example, in our running example, we handled the conflict of the enabling/disabling of the battery service and the enabling/disabling of the driver's door open service.

Although the elicitation of requirements is not in the (main) focus of this thesis, we have seen that the *detection of missing requirements* (e. g. due to the information required by parameter to service relationships) is also a nice side-effect of our methodology. For example, the XOR relationship requires the information what has to be done in case a service is already running and another service is called.

In our approach, we have a quite *seamless transition from informal service descriptions to formal service models*. This ensures that people of different formal backgrounds are served. For example, the client can reason about the system specification given by the service graph and the informal textual service descriptions behind the inner nodes and the leaves. The designer can reason about the system specification given by the formal model. Furthermore, we gave guidelines how to informally specify functional requirements to ensure a smooth transition to our methodology. This helps even more in bridging the gap between informally specified and formally specified requirements.

Looking at different requirements specifications in industry, we observed that the identification and specification of the *system boundaries* is often neglected. In most cases only signal/message names were listed as inputs and outputs. In our methodology we take care of this fact by introducing a dedicated step: the identification of the (logical) system interface (see Section 4.5, *Logical syntactic system interface*). Thus it is specified which information is needed by the system (e. g. the current seat position and the velocity of the car) and what quantities are controlled by the system. The inputs and outputs are described on a logical level. This increases the understandability.

As another consequence of the modular approach, parts of the system functionality can be specified in isolation and integrated afterward. This allows for *distributed development*. This can for example be observed in 4.8.4 (*Combination of the services* FRONT

DOOR OPEN, MANUAL ADJUSTMENT, ERR LOW BATTERY MANUAL, ADJUSTMENT BY MEMORY *and* ERR HIGH VELOCITY).

Due to the modular character of the approach, the model can be modified quite easily (*enhanced modifiability*). For example, if relationships between services change, the modular service specification does not have to be changed. The stepwise modification of the service specification might have to be redone and adjusted. If a service relationship changes, this might only cause effects to the relationship service which realizes the service relationship. Simplified spoken, the advantage of our approach is that the information is specified where needed. For example, the execution status of a service is specified within the service. The affection of a service behavior (e. g. when the service is interrupted) is specified in the service behavior. Conflicts can be solved modularly, too, by means of conflict solving services.

As mentioned in 4.8.1 (*Concepts*), the status actions introduced (ACTIVE and INACTIVE) might not be sufficient. For example, for some systems it might be necessary to deliver the status "stand-by" or something as some relationships may depend on it. However, our approach can be enhanced pretty easily. The set of status actions can be enlarged by an action STAND-BY for example that is sent to a service realizing a relationship that depends on this status. To that end it should be investigated whether the set of necessary status actions depends on the system type and/or the domain.

As far as the development process is concerned, the design of the system functionality from a black-box perspective of course is just the beginning. Subsequent phases would deal with the white box functionality (see Section 7.3, *Outlook*). The advantage of our model is that it can be distributed as it is comprised of modular services (represented by SSDs). The automata can be distributed to components. This would not be possible, if we specified the overall system functionality by means of one overall automata for example. Therefore, the result of our methodology serves as a *good basis for the subsequent phases*.

As the introduced concepts all are *formally founded*, they are given a precise semantics. For example it is specified what the difference between "disable" and "interrupt" is. Additionally, the process can be tool-supported.

As far as the notational techniques are concerned, suggestions are made for each step. At places where possible, standard notations (like state charts) are used. Additionally introduced notational techniques are very intuitive and easy to learn. However, the methodology does not depend on particular notional techniques. For example, for the formal specification of the services, sequence diagrams like MSCs [ITU-T, 1996] could be used to. In general, arbitrary suitable notational techniques can be used as long as the concepts (e. g. realization of basic relationships) are mapped to them.

7.2.2. Disadvantages of the approach

Despite its advantages, the approach also has disadvantages.

For small scale systems (with trivial services) the approach most probably can be considered to be an "overkill". Therefore, it only seems to be suited for medium to large scale systems. Furthermore, the specification of trivial services like the ERROR LOW BATTERY services might not seem to be appropriate. This in deed makes only sense if service relationships point from or at these trivial services. Otherwise, a leaner model could be achieved by combining the behavior of trivial services with other services. (See also the discussion on an adequate service granularity below.)

In the introduction we motivated a strict black box view onto the system as appropriate for the requirements engineering phase. However, this causes an additional effort when modeling the system as the white box functionality also has to be modeled. Therefore, this strict distinction might not be reasonable for each system. For example for systems which do not exhibit (a high degree of) feature interaction, it might be more advisable to directly model the white box functionality of the system-to-be. Generally speaking the cost performance ratio should be pre-estimated first.

Another disadvantage of our methodology arises because of the strict black box view onto the system functionality. In case services encapsulate a larger part of identical behavior this behavior has to be modeled twice. When modeling the white box behavior of a system, this common behavior could be realized in a separate function which is called by both services. However, service calls are not allowed in our methodology as they already concern the internal realization of the functionality.

As mentioned in the introduction, our methodology is only adequate for modeling the behavior of systems having a trivial set of persistent data. The approach has to be modified accordingly to handle systems with a more comprehensive set of persistent data.

As already discussed, Figure 4.34 visualizes another problem of our methodology. After having sent the respective action to the motor controller the service sends the status action INACTIVE to the XOR relationship service. This however assumes that the controlling of the motor only needs one tick. If it takes longer this has to be taken into account in the specification of the ADJUSTMENT OF BACK BACKWARDS and ADJUSTMENT OF BACK BACKWARDS services. Thus, our methodology either has to make restrictions to the environment or has to obey restrictions of the environment.

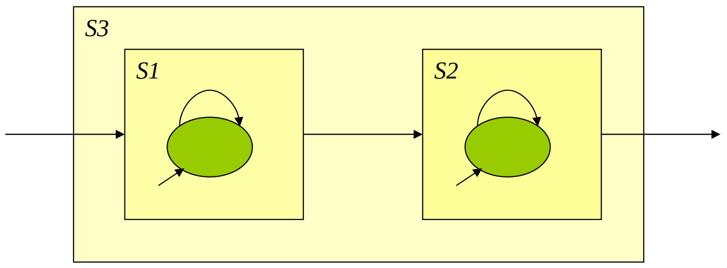

Figure 7.2.: Time tick semantics of SSDs and STDs (SSD + STD)

Time semantics

The worst problems - due to our opinion - arise because of the time tick semantics of our approach. Quantitative timing information is intricate to model. Consider a requirement which demands that the system has to react within five seconds. In this case we could determine that one tick is one second. However, the system's reaction

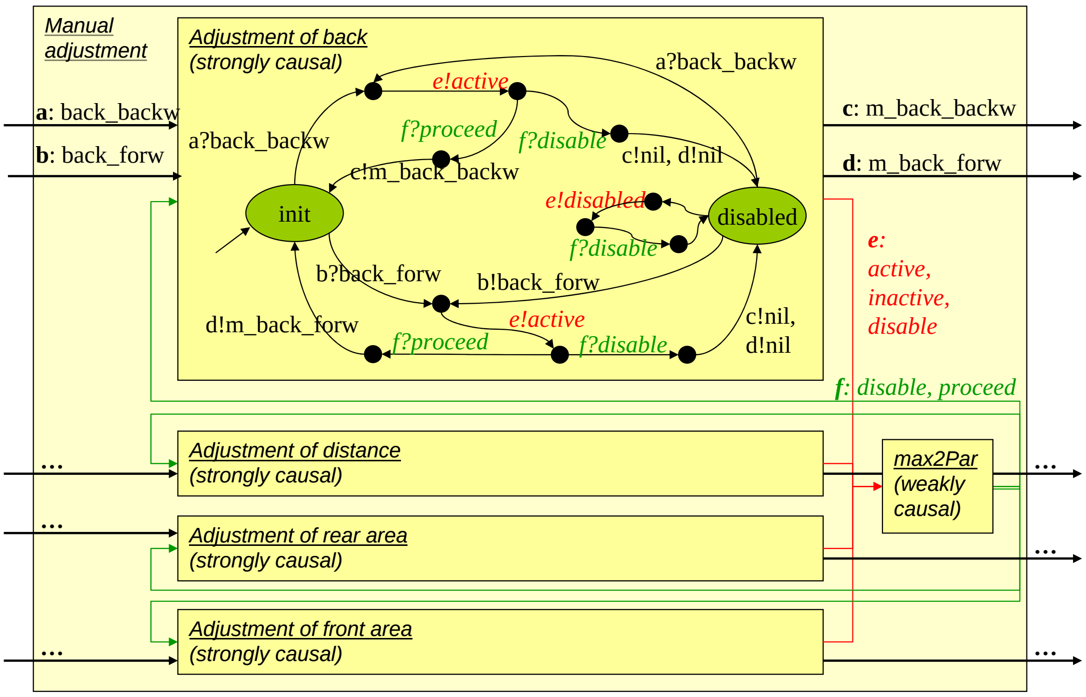

Figure 7.3.: The MANUAL ADJUSTMENT service - modified (SSD + STD)

could occur after one tick (i. e. one second), two ticks, or up to five time ticks. We would have to model all possible behaviors.

Furthermore, according to the semantics of our notational techniques, we introduce a time delay each time we decompose a service. Consider Figure 7.2. Each sub service needs (at least) one time tick to transform input into output. Consequently the super service S3 needs at least two time ticks. When combining sub services to more comprehensive services, these delays sum up. As other services communicating with S3 need the output of S3 their behavior has to be scaled accordingly. In the following we outline how the approach can be modified to handle this problem. However, as the modifications make the approach more complex, it has to be decided whether the richer time semantics are worth the complexity.

In the modified approach we allow for both *weakly causal* and *strongly causal* services. Transitions of weakly causal services cause no time delay whereas transitions of strongly causal services take at least one time tick [Broy and Stolen, 2001]. In the latter case, in our methodology, transitions take exactly one time tick. Atomic services are strongly causal entities. Relationship services, status calculating services, basic handling services, and conflict solvers are weakly causal entities. As loops always incorporate an atomic service, we do not get problems with the semantics and thus do not have to introduce delays.

Additionally, each transition of an atomic service is refined into 4 segments. The service behavior for each segment is defined as follows:

1 segment: Read the system inputs (i. e. logical input actions that are visible at the system boundary).

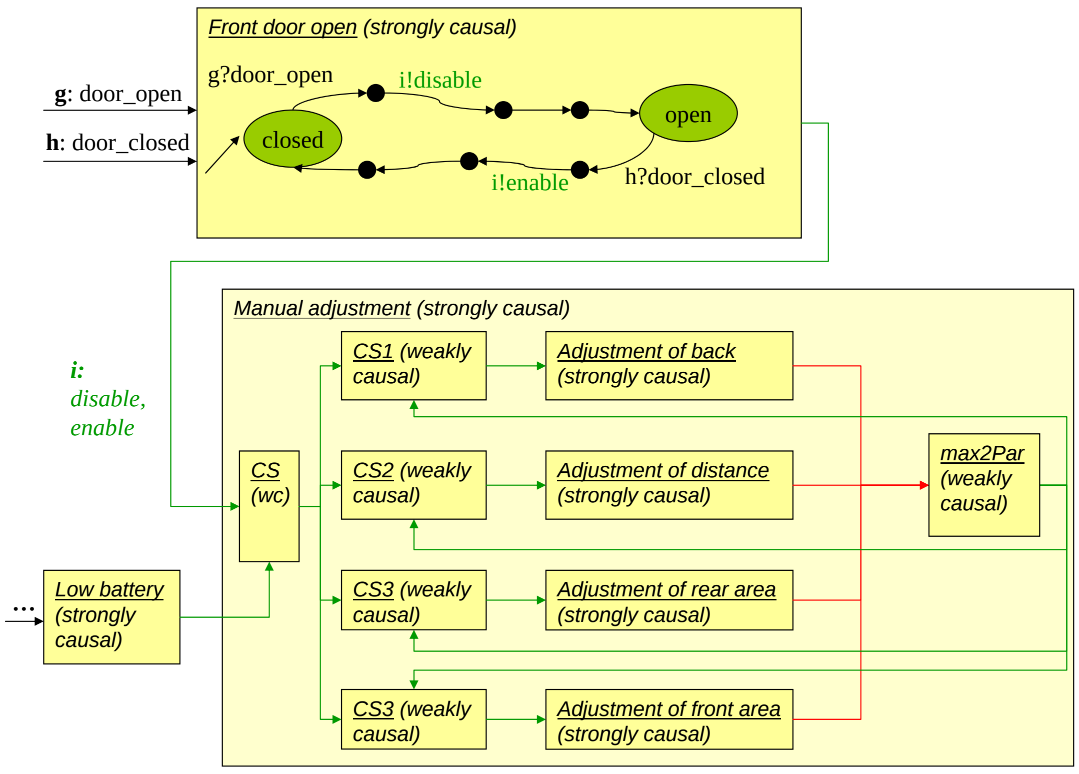
Front door open (strongly causal)
g?door_open
i!disable
open
closed
i!enable
h?door_closed
g: door_open
h: door_closed
i: disable, enable
Manual adjustment (strongly causal)
CS (wc)
CS1 (weakly causal)
CS2 (weakly causal)
CS3 (weakly causal)
CS3 (weakly causal)
Adjustment of back (strongly causal)
Adjustment of distance (strongly causal)
Adjustment of rear area (strongly causal)
Adjustment of front area (strongly causal)
max2Par (weakly causal)
Low battery (strongly causal)
...

Figure 7.4.: The services MANUAL ADJUSTMENT, FRONT DOOR OPEN, and LOW BATTERY - modified (SSD + STD)

2 segment: Send the status that the service is going to be in (ACTIVE or INACTIVE) and the intended ENABLE/DISABLE/INTERRUPT/CONTINUE requests to other services (if need be).

3 segment: Read the ENABLE/DISABLE/INTERRUPT/CONTINUE actions sent by other services (if this is the case).

4 segment: Send the system outputs (i. e. logical output actions that are visible at the system boundary) if not being DISABLED/INTERRUPTED or go to the DISABLED/INTERRUPTED state without sending the system outputs.

The execution of the 4 segments is performed sequentially and takes one time tick altogether.[1]

Figures 7.3 and 7.4 contain an extract of the case study as an example.[2] The transitions of the service ADJUSTMENT OF BACK are refined accordingly. If the service is in the initial state, it first reads the input actions and then sends the status it is going to be in - namely ACTIVE. In the next segment, i. e. the third segment, it receives the commands by the weakly causal service relationship MAX2PAR. If the service is allowed to proceed (indicated by receiving the action PROCEED) the service sends its outputs. If not, the service does not send its outputs (but NIL) and enters the state DISABLED. As long as the service is in the state DISABLED it asks the MAX2PAR service for permission to execute its behavior if it received the input actions. If it is allowed to, it behaves as specified modularly. If not, is stays in the state DISABLED.

In Figure 7.4 we see the sequential composition of several conflict solving services. As conflict solvers are weakly causal entities, no time delay is introduced.

As mentioned above, the model now gets more complex. However, the model can be obtained by the modular specification quite easily. For example, one could specify the activity status in a separate table. For each state and each input situation, one could specify the status of the service (i. e. ACTIVE or INACTIVE). As a consequence we get the label of the second segment of each transition. Additionally, in the third segment it is checked whether the service is interrupted or disabled.[3] The service would then enter the DISABLED or INTERRUPTED state, respectively. In the case of interruption, the state which was left has to be remembered, too, to enable the continuation later. In case the service is allowed to proceed, the service sends the outputs as specified modularly. The rest of the behavior can be obtained similarly.

The ideas presented above are described in more detail in [Hölzl and Rittmann, 2008]. The interested reader is referred to this document.

Variation of schematic procedure and service granularity

Another problem is that the consequent composition of (primitive) services may result in models that are more complex than necessary. The presented approach pro-

[1]This proceeding is similar to micro programming where the interrupt line is checked before a command is executed.

[2]For reasons of clarity, the services ADJUSTMENT OF THE BACK BACKWARDS and ADJUSTMENT OF THE BACK FORWARDS are modeled with one STD.

[3]Note that a DISABLE and INTERRUPT command can not be received at the same time as a conflict solving service would already resolve this conflict.

vides a schematic procedure and leads the developer. However, at some points the models get too complex. Of course, if we do not proceed like the methodology suggests we can optimize the models. The questions arise *if* we can diverge from the procedure and if yes *how*.

The first question can be answered quite easily. As our approach is based on the principle of modularity it does not matter how the internal model of a service looks like. The important thing is that the specifications of the atomic or combined services realize the interfaces that are demanded by the service relationships. For example it does not matter how the MANUAL ADJUSTMENT functionality is internally structured and realized as long as it can handle the DISABLE and ENABLE requests and resolve conflicts. However it may also be necessary to add information about the timing behavior of a service (see discussion above).

The second question is more intricate to answer. First, the level of the service granularity is important. If services are decomposed too far, the models get more complex than needed. For example, as far as our running example is concerned, we definitely carried the decomposition too far. However, we wanted to explain all concepts of the approach by means of a well manageable case study. In general it is difficult to answer what the appropriate level of service granularity is. This problem resembles the question when to introduce a separate class in object-oriented programming. As mentioned before, it may be adequate to introduce sub services if relationships only point to or from a sub behavior of a service. Moreover, a complex functionality should be decomposed into sub services in order to handle the complexity. Of course the experience of the developer has to be taken into account, too. If the black box specification of a service is already realized and should be reused, it is not necessary to decompose it either. This is for example possible in an OEM-supplier-relationship where the internal realization of a package is often not disclosed.

Another way to simplify the model is to use high level states in order to realize exclusively available behavior indicated by the XOR service relationship. The high level states would then contain a state for each exclusively executed service and the switch condition (i. e. the XOR logic) on the transitions. However this is only possible if the services are not required to oppose a certain behavior while being disabled/interrupted as the control flow is only given to one service at the time. For example if there exists an XOR service relationship between the services A and B and B is supposed to issue an error message while being disabled, it also needs control flow. Thus, an XOR relationship can not always be realized by introducing a high level automaton.

Furthermore, simple conflicts (e. g. concurrent disable and interrupt requests) can be solved without the introduction of an additional conflict solver. At this place the transitions can be labeled accordingly. For example, a transition with the input pattern A?DISABLE, B?INTERRUPT (PRIO==5) may lead to the state DISABLED and not to the state INTERRUPTED.

To put it in a nutshell, a schematic procedure as given by our methodology is supposed to guide the developer, but may result in too complex models. According to their experience, the developer is free to diverge from the schema and to optimize the model as long as the interfaces demanded by the service relationships are realized properly.

7.2.3. Experimental analysis of the approach

In the last sections we listed advantages and disadvantages of the approach. In order to evaluate it, we give some experimental analysis at this place. We hereby proceed similarly as proposed by the Goal Question Metric approach (GQM) [Basili et al., 1994] which is a measurement mechanism for feedback and evaluation. We first identify the overall goal of our methodology, derive questions to characterize the abstract goal, and give metrics on how to assess the degree of achievement. Finally we describe the necessary information and give the results of the analysis.

The overall goal of the methodology is to *improve the quality of the requirements engineering process from all stakeholders' perspectives*[4]. In order to determine how well this goal is achieved, many questions are relevant. In the following we give and answer those questions which we consider to be most important:

1. How well do stakeholders (with different backgrounds) understand the notational techniques used?
2. Is the quality of the requirements specification document improved by the process?
3. Is there a seamless transition to the design phase?
4. How well are dependencies between system functionalities captured?

Question 1 is important as stakeholders with different backgrounds are involved in the requirements process, e. g. people of the marketing department and designers having an informal and formal background, respectively. Furthermore, for the acceptance of the approach it is also important to have notational techniques which are intuitive.

The requirements specification document is the result of the requirements engineering phase and thus the central artifact. In practice, the quality of the requirements specification document is crucial as incomplete, inconsistent, and imprecise requirements lead to problems in the subsequent development phases. Question 2 is used to assess the quality of the requirements specification document in regard to its degree of completeness, preciseness, and consistency.

Question 3 characterizes a known problem of current development processes. As mentioned in the introduction, there is a gap between the informal requirements engineering phase and the formal design phase. Thus, the transition is error-prone.

In the thesis at hand we focus on multi-functional systems, i. e. systems which are characterized by a high degree of dependencies between functionalities. We thereby also focus on how well relationships are captured by our methodology (see Question 4).

In the following we will answer each question. To that end we give metrics to assess the answers and perform the evaluation, respectively. For a quantitative evaluation of the questions, comprehensive empirical studies would be necessary. As this is not in the scope of this thesis, we give a subjective evaluation which is based on our

[4]In the GQM approach it is explicitly demanded to specify the viewpoint from which the goal is considered. For our evaluation we do not take into account a particular stakeholder, therefore we chose to specify the goal "from all stakeholders' perspectives".

personal experience and on feedback given by colleagues (from academia) and practitioners (from industry) in the course of respective PhD seminars and an ongoing collaboration project with a car manufacturer.

How well do stakeholders (with different backgrounds) understand the notational techniques used?

Due to our experience, the informal service hierarchy and service graph are a good means to discuss the system functionality and the dependencies with people who do not have a formal background. As we make use of a FODA-like tree notation and describe the dependencies informally (by textual descriptions) this informal model was immediately understood. Almost no questions came up when presenting these informal models.

The formal models (i. e. System Structure Diagrams and State Transition Diagrams) were not immediately understood by people with informal background. However, this was not stringently necessary as the system functionality was already discussed with the help of the informal models. People having a formal background (like designers) understood the models with no problems as SSDs and STDs are just dialects of known notational techniques like structure diagrams and automata. We basically had to describe the underlying semantics of the notational techniques.

However, there was one problem which we had to deal with: Although people were convinced why a pure black box view onto the system functionality (without information about the realization) is important, it was is hard to make people actually stick to this outer perspective. Often people wanted to decompose services into functions that are called by other services and to introduce call relationships. Additionally, practitioners did not abstract from technical signals, but continued to use very technical messages/signals for the description of the system boundaries.

Is the quality of the requirements specification document improved by the process?

As mentioned above, the requirements specification document is the central result of the requirements engineering phase. It is important that the requirements are described *completely, consistently,* and *without ambiguities* in order to avoid errors in the subsequent phases. In this paragraph we describe our experiences concerning which errors where found in an existing requirements specification document when applying our approach to it.

To that end, we modeled the black box functionality of the power seat control system which is described in [Houdek and Paech, 2002]. [Houdek and Paech, 2002] is an exemplary requirements specification document for a fictitious automotive door control unit. It is not a real industrial specification, however practitioners from a car manufacturer were engaged in its creation. It aims at being the basis for case studies and claims to match industrial specifications regarding content and complexity. We chose this case study as due to issues of secrecy, it is hard to get real requirements specification documents from industry. Furthermore, we did not want to create our own requirements specification document (as a basis for the evaluation) as we are familiar with our approach and already pay attention to critical issues like the specification of dependencies between services. Moreover we assume that the specification had been reviewed (prior to publication) and thus serves as a realistic basis.

We applied our approach to the power seat control system of [Houdek and Paech, 2002]. Additionally, we totalized the model to obtain a complete functionality. No tool support was available therefore the models (service hierarchy/graph, SSDs, and STDs) were drawn on paper. Every time missing, imprecise, or conflicting requirements were identified, the type of error was classified and its occurrence was counted. Furthermore, we introduced missing requirements and resolved conflicts due to our idea of how the system functionality is reasonable as no other stakeholders were available.

After having modeled the exemplary power seat control system we obtained the following evaluation:

- Amount of requirements to be modeled: 20[5]
- Missing requirements: 7
- Imprecise requirements: 1 (we did not understand what the requirement demands)
- Miscellaneous: 2 requirements demanding the same functionality (thus 1 requirement is redundant)

As far as the missing requirements are concerned, one missing requirement was needed to describe a missing relationship parameter. It was determined - as in our running example - that at most two directions of the seat are allowed to be controlled at the same time. However it was not specified, what happens if more than two directions are called at the same time. Two further requirements had to be added to describe missing service relationships. We identified two disable relationships but no enable relationships for two services, respectively. Moreover, three requirements were needed to describe the exact service behaviors. It was specified that an error message has to be processed but not if this error message shall be processed once or continuously as long as the error holds. Last, the initial values for the seat position were not given. A requirement demanding that the automatic adjustment can only take place after a position has been saved was not given either.

Note, that in order to show that the introduced requirements engineering methodology reaches its aim - the complete, precise, consistent specification of functional requirements - one would have to show that the resulting requirements specification document does not have any more missing or conflicting requirements. However, this can not be accounted as the only way to detect errors is to perform reviews. Therefore, we showed how many missing, imprecise, and conflicting requirements were detected.

As a result of our experiment we also see another advantage: Quality checks for a requirement specification document can be automated. For example, it can be checked if there are still underspecified automata missing transitions or exhibiting non-determinism. As the semantics of our notational techniques are formally

[5]In the original specification, some requirements could be split in two or more sub requirements. We structured the requirements as we think is appropriate.

founded, the models already specify the functional requirements precisely.

Is there a seamless transition to the design phase?

Practice often suffers from the gap between the informal, textual descriptions of the requirements engineering phase and the formal models of the design phase. Our methodology aims at bridging the gap by formalizing functional requirements step by step. Concerning the question, whether the transition between these phases is seamless we can only give a subjective evaluation.

Usually, we have textually formulated requirements at the end of the requirements phase. In our methodology, the result is a formal model of the system functionality. The result is modeled by typical design models, i. e. structure diagrams (SSDs) and an automata dialect (STD). To answer the question above we have to investigate if the result of our approach matches the starting point of the subsequent design phase. This is difficult to answer as industry usually (and unfortunately) does not make use of design models but immediately starts with coding. Often the requirements documents are generated afterward on basis of the implementation. With regard to an "ideal" development, the result of our approach seems to be a good starting point for design.

We further experienced the following fact: The better the structure of the informal requirements specification is, the easier it is to develop the service graph. In requirements documents, usually one functionality is described per section or subsection (depending on how comprehensive the functionality is). Usually, the relationships between functionalities are specified in either of the sections describing the functionalities. Sometimes dependencies are also specified in the common super section. If there exist dependencies between a functionality and other functionality that is specified several pages later in the document, the handling of these dependencies is error-prone. Usually these relationships are only taken into account when designing the functionality where the dependency is mentioned. Of course, the opposite also holds: The worse a requirements specification document is structured, the more effort it takes to obtain the service graph.

How well are dependencies between system functionalities captured?

Another goal of our approach is to make dependencies between functionalities explicit and to understand their effects. In order to assess if this goal is met, we again, can only give a subjective evaluation: The service hierarchy explicitly captures relationships between user observable services. The effects of these relationships are specified by the systematic adaption of the modular service specifications.

Due to feedback by our industrial partner, relationships like "disable" and "interrupt" (as defined in this thesis) are often confused in practice. Furthermore, practitioners with who we spoke especially appreciated the fact that our methodology also investigates dependencies between relationships (like the enabling/disabling of the battery and the front door open services).

Further remarks and limitation of study

Tool support is inevitable. As there is no tool support yet (see Section 7.3, *Outlook*) we had to make use of other tools (like the organigrams provided by Microsoft Office Visio) or even paper and pencil to draw the service hierarchies/graphs of the case studies. The tool AutoFOCUS 2 is suitable for the specification of the logical architecture (hierarchical SSDs with assigned STDs). However, the stepwise creation of hierarchical models out of already specified models is difficult. Additionally, we faced the above mentioned problems concerning the time ticks. However, we made good experiences with the consistency checks and possibility of simulation provided by AutoFOCUS 2.

After a PhD seminar at university, two colleagues showed their interest in our work. They are concerned with testing human user interfaces and might base their work on our (extended) set of basic service relationships.

The experience presented in this section is just a first step toward a comprehensive evaluation.[6] For a detailed evaluation the following limitations of study have to be overcome:

First of all, a more comprehensive system should be modeled. This system should be a real, industrial system. However, as mentioned above, it is difficult to get such sensitive data as they are competitive advantage.

For evaluating the effects of the new requirements engineering approach, it has to be embedded in an overall process. Errors are the more expensive the later they are found in the development process. Therefore, errors made in the requirements engineering phase can cause high costs. In order to calculate the cost performance of our approach it would be necessary to consider the overall process. Only then, the additional effort in modeling the requirements can be judged and justified.

Another limitation of the study is that the approach was applied by the inventor herself. To get an understanding of how easy the approach is to learn, typical requirements engineers from industry should apply and evaluate it.

Another interesting point is the question how good the approach could be used in industry in general. This is concerned with a migration strategy of the academic approach to an industrial setting. As mentioned, industry rarely uses models. Usually, they immediately start with the coding. Within the ongoing collaboration project with a car manufacturer, we intend to make first steps to realize ideas of the approach in industry. For example, the industry - as a first step - could make use of service graphs perhaps annotated with (semi-) formal specifications like UML Use Case Diagrams or Sequence Diagrams. Seminars with practitioners are planned which will give insights on how this can be done and further potential for the realization of the approach in a non-academic surrounding.

The validation given in this section is just a first step. By applying the approach to a small to medium-scale case study, a good impression on the advantages can be achieved. However, we did not identify and discuss the costs that arise due to the application of the approach. In order to evaluate whether it is advisable to make

[6] As mentioned above, we will evaluate this work further during an ongoing project. Due to reasons of secrecy the results will probably not be made public.

use of an approach, its cost-performance ratio has to be calculated. This however is intricate. As a next step it would furthermore be necessary to compare the cost-performance ratio of our approach with the cost-performance ratio of alternative approaches. This is again difficult as the cost-performance ratio of other approaches of course has to be known but is rarely documented or not documented at all in literature. Furthermore, if the advantages of two approaches diverge, the comparison is even more difficult.

In summary, analyses of the cost-performance ratio and a comparison with cost-performance ratios of alternative approaches are necessary to decide whether the application of the approach is profitable.

7.3. Outlook

In this thesis we presented a methodology for modeling the usage behavior of a multi-functional system. We hereby only modeled its black box functionality, i. e. the behavior that can be observed at the system boundaries. This is adequate for the requirements engineering phase which aims at describing what the system has to do and not at describing already the technical realization. For an overall development methodology of course it has to be investigated how to proceed from the result of our approach. Future research topics include the transition from a model of the black box functionality to a model of the *white box functionality*. Further topics deal with the mapping of the system functionality to a (distributed) component architecture.

In Section 4.10 (*Further considerations*) we already mentioned interesting questions related to our approach. For example the introduction of *different views* onto the (possibly very comprehensive) service graph and *dependency analyses*.

Moreover, it has to be investigated how *verification* of the formal model of the system functionality can be performed, e. g. by model checking.

In our methodology we only modeled functional requirements that contained qualitative timing information (which is the result of causality). Often, however, quantitative timing information is given. The methodology has to be enriched by concepts and notational techniques for dealing with explicit timing requirements or *non-functional requirements* in general. For example the usage of timed automata could be thought of.

The requirements of our running example only describe discrete behavior. Continuous functionality is not covered. The approach also has to be adapted to fit the requirements for modeling *continuous behavior*.

Product line development is an upcoming paradigm which aims at maximizing reuse. The main idea is to explicitly identify commonalities and differences of different products. On basis of a common platform, individual products are developed by adding "building blocks" to this platform. For future work we suggest to enrich our methodology with product line concepts.

As notational techniques for the formal specification of the system services we suggested an automata dialect. It should be investigated if *other notational techniques* are more suitable. This may depend on the system type. For example, for business information systems (BIS), sequence diagrams could be more appropriate as BIS usually are described by interaction sequences. The concepts of the presented approach would then have to be mapped to the other notational technique.

There does not exist a 1:1 mapping between functional requirements and services. As far as the running example of this thesis is concerned, the identification of services was quite easy. However, for other systems this may be quite tricky. Often the system functionality is described by different use cases. The set of use cases implicitly described system services. How to *identify services out of a set of overlapping use cases* definitely is an interesting topic for future work.

Another interesting question is how the formal model of atomic services can be obtained. The transition between informally specified requirements and formal models (automata for example) could be simplified by defining *textual patterns* which can be transformed into models quite easily. This idea is currently investigated in a PhD Thesis (Andreas Fleischmann).

Medium to large scale systems can not be developed at once but have to be decomposed in order to handle complexity. The "pieces" (components) into which the system is composed have to be integrated in the end to obtain the overall system. Today, the decomposition and integration of comprehensive systems is not sufficiently understood. As far as our approach is concerned the following steps seem to be necessary: First, the black box functionality of the system has to be modeled (as presented in this thesis). Then the transition from black box to white box functionality has to be done. The white box functionality can be decomposed into smaller sub systems which in turn are described by a black box view onto the sub system functionality. These sub systems would then be developed (and further decomposed if necessary) and integrated in the end. The *transition from systems to sub systems* is definitely a topic for future research.

In this thesis we introduced a set of basic service relationships. The complex horizontal service relationships can be put down to these basic service relationships. It is probable that for each domain a set of horizontal service relationships could be constant. However, this seems to be an issue of practice instead of future research.

The model of the black box functionality has to be totalized to define the system behavior in each situation. However, some input combinations might not be of relevance as inputs can exclude each other (e. g. the driver can not move the window up and down at the same time). In order to reduce the effort when making the model total, dependencies between logical actions should be captured and taken into consideration (see Section 4.5.1, *Concepts*).

As mentioned in Section 3.2 (*State Transition Diagrams (STDs)*) transitions (within automata) take one time tick. This may impose unwanted time delays. A possibility to avoid this problem is to scale the time ticks within service specifications. How to deal with *time delays* is definitely an interesting topic for future research.

For a pragmatic approach, *tool support* is inevitable. Future work should therefore deal with the development of adequate tools realizing the methodology.

In Section 7.2.3 (*Experimental analysis of the approach*) we gave some experimental analysis based on our experience. Further analysis should be performed to evaluate the approach in more detail. As mentioned above, the approach should be evaluated in the context of an overall development process to analyze the effects the approach also has on other phases of the development process. This is partly done in an ongoing collaboration with industry.

Bibliography

[Abbott, 1983] Abbott, R. J. (1983). Program design by informal english descriptions. *Commun. ACM*, 26(11):882–894. (cited on p 15)

[Adersberger, 2006] Adersberger, J. (2006). FODA, FORM, FOPLE - Supporting material for the course "Produktlinien für Software und Systementwicklung" (Technische Universität München). As of 15.03.2007, `http://www4.in.tum.de/lehre/seminare/hs/SS06/produktlinien/index.shtml`. (cited on p 137)

[Aggoun and Combes, 1997] Aggoun, I. and Combes, P. (1997). Obervers in the SCE and SEE to detect and resolve feature interactions. In *Feature Interactions in Telecommunication Networks IV*, pages 198–212. IOS Press. (cited on p 17)

[Alur and Dil, 1994] Alur, R. and Dil, D. L. (1994). A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235. As of 01.03.2007, `http://citeseer.ist.psu.edu/alur94theory.html`. (cited on p 27)

[Andre, 1996] Andre, C. (1996). Syncharts: A visual representation of reactive behaviors. (cited on p 27)

[AutoRAID, 2007] AutoRAID, T. U. M. (2007). Homepage of the tool AutoRAID. As of 2007, `http://wwwbroy.in.tum.de/~autoraid/`. (cited on pp 14, 16, 130)

[Basili et al., 1994] Basili, V. R., Caldiera, G., and Rombach, D. H. (1994). *Encyclopedia of Software Engineering*, volume 1, chapter The goal question metric approach, pages 528–532. John Wiley & Sons Inc. (cited on p 157)

[Bauer et al., 2005] Bauer, A., Romberg, J., Schätz, B., Braun, P., Ulrich, F., Mai, P., and Ziegenbein, D. (2005). Incremental development for automotive software in AutoMoDe. In *Proceedings of the 3rd Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER3)*, volume 191 of *HNI - Verlagsschriftenreihe*, Paderborn, Germany. (cited on p 130)

[Böckle et al., 2004] Böckle, G., Knauber, P., Pohl, K., and Schmid, K. (2004). *Software-Produktlinien*. Number ISBN: 3-89864-257-7. dpunkt.verlag. (cited on p 11)

[Braithwaite and Atlee, 1994] Braithwaite, K. and Atlee, J. (1994). Towards automated detection of feature interactions. (cited on p 17)

[Broy, 2005] Broy, M. (2005). Service-oriented systems eingineering: Specification and design of services and layered architectures - the JANUS approach. *Engineering Theories of Software Intensive Systems*, pages 47–81. Springer Verlag. (cited on pp 143, 149, 175, 177, 181)

[Broy, 2007] Broy, M. (2007). A theory for requirements specification and architecture design of multi-functional software systems. Unpublished. (cited on pp 16, 64, 65, 76, 83, 176, 182, 183)

[Broy et al., 2007] Broy, M., Krüger, I. H., and Meisinger, M. (2007). A formal model of services. *ACM Transactions on Software Engineering Methodology (TOSEM)*, 16(1). (cited on p 5)

[Broy and Steinbrüggen, 2004] Broy, M. and Steinbrüggen, R. (2004). *Modellbildung in der Informatik*. Number ISBN 1439-5428. Springer. (cited on p 10)

[Broy and Stolen, 2001] Broy, M. and Stolen, K. (2001). *Specification and Development of interactive systems - FOCUS on Streams, Interfaces, and Refinement*. Number ISBN 0-387-95073-7 in Monographs in Computer Science. Springer. (cited on pp 28, 40, 143, 149, 153, 178, 180, 183)

[Buschmann et al., 1996] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). *A System of Patterns. Pattern-Oriented Software Architecture*. Wiley. (cited on p 137)

[Calder et al., 2003] Calder, M., Magill, E., Kolberg, M., and S., R.-M. (2003). Feature interaction: A critical review and considered forecast. *Computer Networks*, 41/1:115–141. Available at: http://www.dcs.gla.ac.uk/~muffy/papers/calder-kolberg-magill-reiff.pdf. (cited on p 17)

[Clements and Northrop, 2002] Clements, P. and Northrop, L. (2002). *Software Product Lines: Practices and Patterns*. Number ISBN-10: 0-201-70332-7 in The SEI Series in Software Engineering. Addison-Wesley. (cited on p 11)

[CMU, 2007] CMU (2007). Homepage of the Carneggie Mellon University (CMU), information about software product lines. As of 15.03.2007, http://www.sei.cmu.edu/productlines/. (cited on p 138)

[Czarnecki, 1998] Czarnecki, K. (1998). *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models*. PhD thesis, Technische Universität Ilmenau. (cited on p 18)

[Damas et al., 2006] Damas, C., Lambeau, B., and van Lamsweerde, A. (2006). Scenarios, goals, and state machines: a win-win partnership for model synthesis. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 197–207, New York, NY, USA. ACM. (cited on p 15)

[Damm and Harel, 2001] Damm, W. and Harel, D. (2001). LSCs: Breathing life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45–80. (cited on p 14)

[Dano et al., 1997] Dano, B., Briand, H., and Barbier, F. (1997). An approach based on the concept of use case to produce dynamic object-oriented specifications. In *RE '97: Proceedings of the 3rd IEEE International Symposium on Requirements Engineering (RE'97)*, page 54, Washington, DC, USA. IEEE Computer Society. (cited on p 14)

[Darimont et al., 1997] Darimont, R., Delor, E., Massonet, P., and van Lamsweerde, A. (1997). GRAIL/KAOS: an environment for goal-driven requirements engineering. In *ICSE '97: Proceedings of the 19th international conference on Software engineering*, pages 612–613, New York, NY, USA. ACM. (cited on p 14)

[Davis, 1982] Davis, A. M. (1982). The design of a family of application-oriented requirements languages. *Computer*, 15 (5):21–28. (cited on p vii)

[de Alfaro and Henzinger, 2001] de Alfaro, L. and Henzinger, T. A. (2001). Interface automata. *SIGSOFT Softw. Eng. Notes*, 26(5):109–120. (cited on p 27)

[Deubler, 200x] Deubler, M. (200x). *Strukturierte Nutzungssicht für multifunktionale Systeme*. PhD thesis, Technische Universität München. To appear. (cited on p 16)

[Deubler et al., 2005] Deubler, M., Grünbauer, J., Holzbach, A., Popp, G., and Wimmel, G. (2005). Kontextadaptivität in dienstbasierten Softwaresystemen. Technical Report TUM-

I0511, Institut für Informatik, Technische Universität München TUM. (cited on p 140)

[Deubler et al., 2004a] Deubler, M., Grünbauer, J., Jürjens, J., and Wimmel, G. (2004a). Sound development of secure service based systems. In *Proceedings of the 2nd International Conference on Service Oriented Computing (ICSOC)*. (cited on p 140)

[Deubler et al., 2004b] Deubler, M., Grünbauer, J., Popp, G., Wimmel, G., and Salzmann, C. (2004b). Tool supported development of service-based systems. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference (ASPEC)*, pages 99–108, Busan, Korea. IEEE Computer Society. (cited on pp 3, 140)

[Deubler et al., 2004c] Deubler, M., Grünbauer, J., Popp, G., Wimmel, G., and Salzmann, C. (2004c). Towards a model-based and incremental development process for service-based systems. In *Proceedings of the IASTED International Conference on Software Engineering (IASTED SE 2004)*. As of 04.07.2007, http://www4.in.tum.de/~gruenbau/. (cited on pp xv, 140, 141)

[Dijkstra, 1972] Dijkstra, E. W. (1972). Notes on structured programming. In *Structured Programming*, Academic Press New York. Dahl, O.-J. and Hoare, C. A. R. and Dijkstra, E. W. (cited on p 2)

[Faulk et al., 1992] Faulk, S., Brackett, J., Ward, P., and Kirby, J. J. (1992). The core method for real-time requirements. *IEEE Software*, pages 22–33. (cited on p 143)

[Felty and Namjoshi, 2000] Felty, A. P. and Namjoshi, K. S. (2000). Feature specification and automated conflict detection. In *Feature Interactions Workshop*. IOS Press. (cited on p 17)

[Fey et al., 2002] Fey, D., Fatja, R., and Boros, A. (2002). Feature modeling: A meta-model to enhance usability and usefullness. In *Software Product Lines : Second International Conference (SPLC 2)*, pages 198–216. (cited on p 18)

[Fillmore, 1971] Fillmore, C. J. (1971). *Universals in Linguistic Theory*, chapter The case for the case, pages 1–118. (cited on p 15)

[Filman et al., 2004] Filman, R., Elrad, T., Clarke, S., and Aksit, M. (2004). *Aspect-Oriented Software Development*. Addision Wesley Professional. (cited on p 5)

[Fleischmann et al., 2005] Fleischmann, A., Hartmann, J., Pfaller, C., Rappl, M., Rittmann, S., and Wild, D. (2005). Concretization and formalization of requirements for automotive embedded software systems development. In *Proceedings of the 10th Australien Workshop on Requirements Engineering (AWRE)*, Melbourne, Australia. (cited on p 2)

[Frischkorn, 2004] Frischkorn, H.-G. (2004). Automotive software - the silent revolution. In *Proceedings of the Automotive Software Workshop San Diego (ASWSD 2004)*. Abstract and Presentation. (cited on p 1)

[Fuchs and Mendler, 1994] Fuchs, M. and Mendler, M. (1994). *Formal Semantics for VHDL*, chapter Functional Semantics for Delta-Delay VHDL based on Focus, pages 9–38. Kluwer Academic Publishers. (cited on p 32)

[Geisberger and Schätz, 2007] Geisberger, E. and Schätz, B. (2007). Modellbasierte Anforderungsanalyse mit AutoRAID. *GI - Informatik Forschung und Entwicklung*. (cited on pp xv, 131)

[Gery et al., 2002] Gery, E., Harel, D., and Palachi, E. (2002). Rhapsody: A complete lifecycle model-based development system. In *Integrated Formal Methods: Third International Conference, IFM 2002, Turku, Finland, May 15-17, 2002. Proceedings*, volume 2335/2002 of *Lecture Notes in Computer Science*. (cited on p 130)

[Goldin and Berry, 1997] Goldin, L. and Berry, D. (1997). Abstfinder, a prototype natural language text abstraction finder for use in requirements elicitation. (cited on p 15)

[Gomaa, 1984] Gomaa, H. (1984). A software design method for real-time systems. In *Com-*

munications of the ACM, number 27(9), pages 938–949. (cited on p 139)

[Graf and Hooman, 2004] Graf, S. and Hooman, J. (2004). *Software Architecture*, volume 3047/2004 of *Lecture Notes in Computer Science*, chapter Correct Development of Embedded Systems, pages 241–249. Springer Berlin / Heidelberg. (cited on p 130)

[Grosu et al., 1996] Grosu, R., Klein, C., Rumpe, B., and Broy, M. (1996). State transition diagrams. Technical Report TUM-I9630, Technische Universität München. (cited on p 27)

[Gurevich, 2000] Gurevich, Y. (2000). Abstract state machines - theory and applications. In *Proceedings of the international workshop on abstract state machines*, number 1912 in Lecture Notes in Computer Science. (cited on p 27)

[Harel, 1987] Harel, D. (1987). Statecharts: A visual formalism for complex systems. In *Science of Computer Programming*, volume 8, pages 231–274. (cited on p 29)

[Hartmann et al., 2006a] Hartmann, J., Fleischmann, A., Pfaller, C., Rappl, M., Rittmann, S., and Wild, D. (2006a). Feature Net - ein Ansatz zur Modellierung von automobil- spezifischem Domänenwissen und Anforderungen. In *Proceedings of the 4th Workshop on Automotive Software Engineering (ASE 2006) held in conjunction with the annual congress of the "Gesellschaft für Informatik"*, Dresden Germany. (cited on p 2)

[Hartmann and Harhurin, 2008] Hartmann, J. and Harhurin, A. (2008). A formal approach to specifying the functionality of sofware system families. In *Submitted as contribution to the conference on Fundamenal Approaches to Software Engineering 2008 (FASE 2008)*. (cited on p 185)

[Hartmann et al., 2006b] Hartmann, J., Rittmann, S., Scholz, P., and Wild, D. (2006b). A compositional approach for functional requirement specifications of automotive software systems. In *Proceedings of the Workshop on Automotive Requirements Engineering (AuRE 06)*, Minneaplis/St. Pauls, U.S.A. (cited on p 2)

[Hartmann et al., 2006c] Hartmann, J., Rittmann, S., Scholz, P., and Wild, D. (2006c). Formal incremental requirements specification of service-oriented automotive software systems. In *Inproceedings of the Second International Symposium on Service Oriented System Engineering (SOSE 2006)*, Shanghai, China. (cited on p 2)

[Heitmeyer, 2002] Heitmeyer, C. L. (2002). Software cost reduction. In Marciniak, J. J., editor, *Encyclopedia of Software Engineering*, number ISBN: 0-471-02895-9. John Wiley & Sons, Inc., New York, second edition. Available at: http://chacs.nrl.navy.mil/personnel/heitmeyer.html. (cited on p 143)

[Hölzl and Rittmann, 2008] Hölzl, F. and Rittmann, S. (2008). Early simulation of usage behavior of multi-functional systems. Submitted to the Modellierung 2008. (cited on p 155)

[Homayoon and Singh, 1988] Homayoon, S. and Singh, H. (1988). Methods of addressing the interactions of intelligent network services with embedded switch services. *Communications Magazine, IEEE*, 26(12):42–46. (cited on p 17)

[Houdek and Paech, 2002] Houdek, F. and Paech, B. (2002). Das Türsteuergerät - eine Beispielspezifikation. Beispiel-Spezifikation IESE-Report Nr. 002.02/D, Frauenhofer IESE. (cited on pp 21, 23, 46, 110, 158, 159)

[Huber et al., 1997] Huber, F., Schätz, B., and Einert, G. (1997). Consistent graphical specification of distributed systems. volume 1313 of *4th International Symposium of Formal Methods Europe - Lecture Notes in Computer Science*. (cited on pp 16, 26, 27, 28, 29, 32, 77, 143)

[Hwang and Miller, 1995] Hwang, H.-S. and Miller, W. A. (1995). *Computers and Industrial Engineering*, chapter Hybrid blackboard model for feature interactions in process planning, pages 613–617. Elsevier Science. (cited on p 17)

[ITU-T, 1996] ITU-T (1996). ITU-T-recommendations z.120 - Messsage Sequence Chart

(MSC 96). As of 10.12.2006, `http://www.itu.int/ITU-T/studygroups/com10/languages/Z.120_1199.pdf`. (cited on pp 14, 151)

[Jacobson et al., 1997] Jacobson, I., Griss, M., and Jonsson, P. (1997). *Software Reuse: Achitecture, Process and Organization for Business Success*. Number ISBN-10: 0201924765. Addison-Wesley Longman. (cited on p 18)

[Kang et al., 1990] Kang, K., Cohen, S., Hess, J., Novak, W., and Peterson, A. (1990). Feature-oriented domain analysis (FODA) - feasibility study. Technical Report CMU/SEI-90-TR-021, Carnegie Mellon Software Engineering Institute. (cited on pp 18, 137, 138, 139)

[Kang et al., 2002a] Kang, K., Kang, K. C., and Lee, J. (2002a). Concepts and guidelines of feature modeling for product line software engineering. In Gacek, C., editor, *Proceedings of the Seventh Reuse Conference (ICSR7)*, volume 2319 of *LNCS*, pages 62–77, Austin, U.S.A. Springer. (cited on pp 48, 69)

[Kang et al., 1998] Kang, K., Kim, S., Lee, J., Kim, K., Shin, E., and Huh, M. (1998). FORM: A feature-oriented reuse method with domain-specific reference architectures. In *Annals of Software Engineering*, number 5, page 143 168. (cited on pp 18, 138)

[Kang et al., 2002b] Kang, K. C., Lee, J., and Donohoe, P. (2002b). Feature-oriented product line engineering. In *IEEE Software*. IEEE. (cited on pp 18, 138)

[Kelly et al., 1995] Kelly, B., Crowther, M., King, J., Masson, R., and DeLapeyre, J. (1995). Service validation and testing. (cited on p 17)

[Kof, 2001] Kof, L. (2001). Formales Service Engineering für Eingebettete Systeme. Diploma Thesis at Technische Universität München. (cited on p 5)

[Kof, 2005] Kof, L. (2005). *Text Analysis for Requirements Engineering*. PhD thesis, Technische Universität München. (cited on p 15)

[Kof et al., 2004] Kof, L., Schätz, B., Thaler, I., and Wisspeintner, A. (2004). Service-based development of embedded systems. In *Net.Object Days, OOSE Workshop*, Erfurt, Germany. (cited on p 142)

[Krüger, 2002] Krüger, I. H. (2002). Specifying services with UML and UML-RT. In Caillaud, B. and Muscholl, A., editors, *Electronic Notes in Theoretical Computer Science*, volume 65 (7). Elsevier Science B. V. (cited on p 5)

[Kulkarni and Reddy, 2005] Kulkarni, V. and Reddy, S. (2005). *UML Modeling Languages and Applications*, volume 3297/2005 of *Lecture Notes in Computer Science*, chapter Model-Driven Development of Enterprise Applications, pages 118–128. Springer Berlin / Heidelberg. (cited on p 130)

[Lee et al., 2000] Lee, K., Kang, K. C., Koh, E., Chae, W., Kim, B., and Choi, B. W. (2000). Domain-oriented engineering of elevator control software: A product line practice. In Donohoe, P., editor, *Proceedings of theFirstSoftware Product Line Conference*, pages 3–22. (cited on p 18)

[Lorentsen et al., 2001] Lorentsen, L., Tuovinen, A. P., and Xu, J. (2001). Modeling feature interactions in mobile phones. In *ECOOP Workshop - Feature Interaction in Composed Systems*. (cited on p 17)

[Lynch et al., 2003] Lynch, N., Segala, R., and Vaandrager, F. (2003). Hybrid I/O automata. In *Information and Computation*, volume 185, Issue 1, pages 105–157. (cited on p 27)

[Maraninchi, 1991] Maraninchi, F. (1991). The Argos language: Graphical representation of automata and description of reactive systems. In *Proceedings of the IEEE International Conference on Visual Languages*, Kobe, Japan. (cited on p 27)

[Marples and Magill, 1998] Marples, D. and Magill, E. H. (1998). The use of rollback to prevent incorrect operation of features in intelligent network based systems. In *Feature In-*

teractions in Telecommunications and Software Systems, pages 115–135. IOS Press. (cited on p 17)

[Meisinger and Rittmann, 2008] Meisinger, M. and Rittmann, S. (2008). A service comparison. Technical report, Technische Universität München, Munich, Germany. To appear. (cited on pp 5, 47, 135, 175)

[Metzger, 2004] Metzger, A. (2004). *Computer Networks*, volume 45, chapter Feature Interactions in embedded control systems, pages 625–644. Elsevier. (cited on p 17)

[Mobilsoft, 2006] Mobilsoft (2006). Homepage of the mobilsoft project. (cited on p 2)

[Nelson and Prasad, 2003] Nelson, E. C. and Prasad, K. V. (2003). Automotive infotronics: An emerging domain for service-based architecture. In Krüger, I. H., Schätz, B., Broy, M., and Hussmann, H., editors, *SBSE'03 Service-Based Software Engineering - Proceedings of the FM2003 Workshop*, number TUM-INFO-09-I0315-0/1.-FI, pages 3–14. Technische Universität München. (cited on p 1)

[Olderog, 1986] Olderog, E.-R. (1986). Semantics of concurrent processes. volume 29 of *Part I Bulletin of EATCS*, pages 73–97. (cited on p 76)

[OMG, 2003] OMG (2003). Unified Modeling Language: Superstructure. As of 09.10.2007, http://www.omg.org/docs/formal/07-02-03.pdf. (cited on pp 13, 132)

[OMG, 2006] OMG (2006). OMG SysML Specification (final adopted specification). Technical report, OMG. (cited on p 134)

[OMG, 2007a] OMG (2007a). Homepage of Object Management Group. As of 15.03.2007, http://www.omg.org/. (cited on p 132)

[OMG, 2007b] OMG (2007b). Homepage of OMG SysML. As of 15.03.2007, http://www.omgsysml.org/. (cited on p 134)

[oose Innovative Informatik, 2007] oose Innovative Informatik (2007). Homepage of oose Innovative Informatik; site about Systems Modeling Language (SysML). As of 15.03.2007, http://www.oose.de/sysml.htm. (cited on p 134)

[Park, 1976] Park, D. (1976). Finitness is μ-ineffible. *Theoretical Computer Science*, 3(2):173–181. Springer Verlag. (cited on p 32)

[Parnas, 1972] Parnas, D. (1972). On the criteria to be used to decompose systems into modules. *ACM*, 15:1053–1058. (cited on p 2)

[Parnas and Madey, 1995] Parnas, D. L. and Madey, J. (1995). Functional documents for computer systems. *Science of Computer Programming*, 25(1):41–61. (cited on p 143)

[Perng and Chang, 1997] Perng, D.-B. and Chang, C.-F. (1997). *Computer-Aided Design*, volume 29, chapter Resolving feature interactions in 3D part editing, pages 687–699. Elsevier. (cited on p 17)

[Pfaller et al., 2006] Pfaller, C., Fleischmann, A., Hartmann, J., Rappl, M., Rittmann, S., and Wild, D. (2006). On the integration of design and test - a model based approach for embedded systems. In *Proceedings of the Workshop on Automation of Software Test (AST 06)*, Shanghai, China. (cited on p 2)

[Pickett, 2000] Pickett, J. P., editor (2000). *The American Heritage Dictionary*. Houghton Mifflin Company, Boston, MA, USA. (cited on p 139)

[Rechenberg and Pomberger, 1999] Rechenberg, P. and Pomberger, G. (1999). *Informatik-Handbuch*. Number ISBN 3-446-19601-3. Carl Hanser Verlag München Wien. (cited on p 8)

[Reiff, 2000] Reiff, S. (2000). Identifying resolution choices for an online feature manager. In *Feature Interactions in Telecommunications and Software Systems*, pages 113–128. IOS Press.

(cited on p 17)

[Rittmann et al., 2005] Rittmann, S., Fleischmann, A., Hartmann, J., Pfaller, C., Rappl, M., and Wild, D. (2005). Integrating service specifications on different levels of abstraction. In *Proceedings of the IEEE International Workshop on Service-Oriented System Engineering (SOSE)*, Bejing, China. (cited on p 2)

[Rupp, 2007] Rupp, C. (2007). *Requirements Engineering und -Management. Professionelle, iterative Anforderungsanalyse für die Praxis*. Hanser. (cited on p 15)

[Salzmann and Schätz, 2003] Salzmann, C. and Schätz, B. (2003). Service based software specification. In *Proceedings of Intl. International Workshop on Test and Analysis of Component Based Systems (TACOS), ETAPS 2003 Warsaw, Poland 2003*. As of 02.12.2006, http://www4.in.tum.de/%7Esalzmann/salzmannSchaetzFASE03.pdf. (cited on pp 5, 5)

[Schätz, 2002] Schätz, B. (2002). Towards service-based systems engineering: Formalizing and mu-checking service descriptions. Technical Report TUM-INFO-07-I0206-0/1.-FI, Technische Universität München. Available at: http://www4.in.tum.de/~schaetz/public.html (as of 1.12.2006). (cited on p 5)

[Schätz, 2005] Schätz, B. (2005). Building components from functions. In *Proceedings of the International Workshop on Formal Aspects of Component Software (FACS 2005)*, volume 160 of *Electronic Notes in Theoretical Computer Science*. (cited on p 142)

[Schätz, 2007] Schätz, B. (2007). Modular functional descriptions. In *Proceedings of the Fourth International Workshop on Formal Aspects of Component Software (FACS 2007)*, Electronic Notes in Theoretical Computer Science. (cited on pp 16, 34)

[Schätz and Salzmann, 2003] Schätz, B. and Salzmann, C. (2003). Service-based systems engineering: Consistent combination of services. In *Proceedings of ICFEM 2003, Fifth International Conference on Formal Engineering Methods*, number 2885 in LNCS. Springer. (cited on pp 106, 142)

[Shehata et al., 2004] Shehata, M., Eberlein, A., and Fapojuwo, A. (2004). The use of semi-formal methods for detecting requirements interactions. In *Proceedings of the IASTED International Conference on Software Engineering (IASTED SE 2009)*, pages 230–235. (cited on p 17)

[Siedersleben, 2004] Siedersleben, J. (2004). *Moderne Softwarearchitektur - Umsichtig planen, robust bauen mit Quasar*. dpunkt Verlag. (cited on p 139)

[Sommerville, 2004] Sommerville, I. (2004). *Software Engineering*. Number ISBN 0-321-21026-3. Addison Wesley Publishers Limited, Pearson Education Limited, 7th edition. (cited on pp 6, 9, 174)

[TUM, 2006a] TUM (2006a). Homepage of the chair of Software & Systems Engineering, Institut für Informatik, Technische Universtität München. As of 01.02.2006, http://wwwbroy.in.tum.de. (cited on p 130)

[TUM, 2006b] TUM (2006b). Homepage of the project InServe, Technische Universität München. As of 2006, http://www4.in.tum.de/proj/inserve/index.shtml. (cited on p 130)

[TUM, 2006c] TUM (2006c). Homepage of the project MEwaDiS, Technische Universität München. As of 2006, http://www4.in.tum.de/~mewadis/. (cited on pp 130, 140)

[TUM, 2006d] TUM (2006d). Homepage of the tool AutoFOCUS 2. As of 2006, http://wwwbroy.in.tum.de/~af2/. (cited on p 130)

[Turner et al., 1999] Turner, C. R., Fugetta, A., Lavazza, L., and Wolf, A. L. (1999). A conceptual basis for feature engineering. *Journal of Systems and Software*, 49 (1):3–15. (cited on p 5)

[von der Beeck, 1995] von der Beeck, M. (1995). Comparison of statecharts variants. In *Proceedings of the Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT 94)*, number 863 in Lecture Notes in Computer Science, pages 128–148, Lübeck, Germany. Springer. (cited on p 27)

[Wild et al., 2006] Wild, D., Fleischmann, A., Hartmann, J., Pfaller, C., Rappl, M., and Rittmann, S. (2006). An architecture-centric approach towards the construction of dependable automotive software. In *Proceedings of the SAE 2006 World Congress*, Detroit, U.S.A. (cited on p 2)

[Withey, 1994] Withey, J. V. (1994). Implementing model based software engineering in your organization: An approach to domain engineering. Technical Report CMU/SEI-94-TR-01, Carnegie Mellon University, Pittsburgh, Pa, USA. (cited on p 138)

[Zave, 2001] Zave, P. (2001). Feature-oriented description, formal methods, and DFC. In *Language Constructs for Describing Features*, pages 11–26. Springer-Verlag London Ltd. (cited on pp xv, 102, 135, 136, 137)

[Zave, 2003] Zave, P. (2003). An experiment in feature engineering. *Programming methodology*, (ISBN 0-387-95349-3):353–377. Springer-Verlag New York, Inc. (cited on pp 5, 135, 174)

[Zave and Jackson, 2000] Zave, P. and Jackson, M. (2000). New feature interactions in mobile and multimedia telecommunication services. *Feature Interactions in Telecommunications and Software Systems*, pages 51–66. As of 12.06.2006, `http://www.research.att.com/~pamela/fiw6.pdf`. (cited on pp 4, 135)

Glossary

A

atomic service In our approach an atomic service is the smallest piece of (partial) black box functionality which is user visible and likely to be reused. For example, the adjustment of the back backwards is an atomic service, p. 45.

B

basic actions Basic actions realize the internal communication between services realizing a service relationship (→relationship services) and services which are influenced by the relationship. The set of basic actions contains the actions ENABLE, DISABLE, INTERRUPT, CONTINUE, and RESET, p. 88.

basic handling service If a →basic action is sent to a hierarchically decomposed service, the basic handling service forwards the basic actions to the respective sub services, p. 92.

basic service relationship Basic service relationships are the "simplest" →horizontal service relationships that represent the information how a service can affect the black box behavior of another service. The set of basic service relationships is comprised of ENABLE, DISABLE, INTERRUPT, CONTINUE, and RESET. The basic service relationships are realized by sending and receiving the →basic actions, p. 66.

C

combined service In the thesis at hand, combined services are more comprehensive behaviors that contain the behavior of several sub services. All services except the atomic services are combined services. The overall system behavior is the most comprehensive combined service of a system, p. 65.

complex horizontal service relationship Complex horizontal service relationships are all →horizontal service relationships except the →basic service relationships, p. 67.

F

feature interaction For our thesis we adopt the definition of [Zave, 2003] for the term feature interaction: "A feature interaction is some way in which a feature or features modify or influence another feature in defining the overall system behavior". As in our terminology a feature is a service, we also speak of →service interaction in this thesis, p. 5.

functional requirement Functional requirements "are statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations. In some cases, the functional requirements may also explicitly state what the system should not do" [Sommerville, 2004], p. 6.

H

horizontal (service) relationship In our approach we distinguish between →vertical and horizontal (service) relationships. Horizontal relationships describe the effects/dependencies between services, p. 62.

I

influenced service The behavior of a user-visible service can be influenced by another service (e.g. by a relationship service). We call the service the behavior of which is influenced by another service, the influenced service, p. 84.

influencing service The behavior of a user-visible service can be influenced by another service (e.g. by a relationship service). We call the service which influenced the behavior of another service, the influencing service, p. 84.

L

logical action The term logical action refers to the both input actions and output actions. It is an abstract term for message, signal, etc. and abstracts from the technical realization, p. 53.

logical channel Logical channels are used for the communication between services. For each logical action, a logical channel is introduced. Logical channels do not necessarily exist in the later technical realization. The can be understood as logical communication links, p. 54.

logical syntactic interface The logical syntactic interface of a service (or the system under specification) is a listing of all possible input and output channels and their input and output actions, respectively, p. 53.

M

multi-functional system A multi-functional system is characterized by a high degree of interaction / dependencies between (sub-) functions, p. 3.

R

relationship service Horizontal service relationships, e.g. the XOR relationship, are realized by special services which are responsible for realizing the service relationship. They do not have a user-visible behavior, but just realize the service relationship by sending →basic actions to the services which are affected by the relationship. We call these services relationship services, p. 89.

S

service There does not exist a uniform definition for the term service. See [Meisinger and Rittmann, 2008] for an overview of various service notions. In this thesis we adapt the idea of [Broy, 2005] and define a service as follows: A service is a *piece of functionality* - abstracting from technical structure. It is described by a *black box view* relating inputs to outputs and hiding the internal realization. It is a *partial behavior*, i.e. it might not be defined for all possible inputs in each situation. In order to define in which situations the service can be used, and which inputs are needed to guarantee a specific output, a *→service protocol* has to be defined. Quality of service attributes can be annotated to further specify a feature, p. 5.

service graph A service graph is a service hierarchy enriched by the horizontal relationships between the services of the service hierarchy, p. 65.

service hierarchy →Vertical service relationships introduce a hierarchy on services. The root (service) of the hierarchy is the overall system functionality which is comprised of sub-behaviors, p. 64.

service interaction See →feature interaction, p. 5.

service-oriented approach A service-oriented approach is an approach which uses services (in our methodology: pieces of partial functionality) as basic building blocks, p. 5.

service relationship Service relationships make dependencies between services explicit. They contain the effects that services have on each other. We distinguish between →vertical service relationships and horizontal service relationships, p. 62.

standard control interface In order to handle influences by other services, the modular service specification has to be adapted. To that end, a service has to implement so-called standard control interfaces which are specific to the relationships between services, p. 7.

State Transition Diagram (STD) A State Transition Diagram graphically represents the dynamic structure of system entities. In our methodology it is used to describe the behavior of modular services. We also make use of State Transition Diagrams to show how the →basic relationships and the sending of the information about a service's status can be realized schematically, p. 27.

status actions Horizontal service relationships may depend on the execution status of one or more services. The status actions (ACTIVE and INACTIVE) are sent by the user-visible services to the →relationship services in order to provide them with the information of their current status. Note, that the set of status actions can be enlarged if needed, p. 90.

status calculating service A status calculating services calculates the status of a hierarchically decomposed service on basis of the stati of its sub services, p. 92.

sub service →Vertical service relationships introduce a hierarchy on services. The child of a super service (which basically is a more comprehensive behavior) is called a subservice. A formal foundation of the subservice relationship - as used in our approach - can be found in [Broy, 2007], p. 64.

super service →Vertical service relationships introduce a hierarchy on services. The parent of a service (which is basically a more comprehensive behavior) is called its super service. The child of the super service is called a →subservice. The upper most super service is the overall system behavior, p. 64.

system behavior see →system functionality, p. 38.

system functionality The term system functionality refers to the overall functionality provided by the system under specification. It is used synonymously for the →system behavior, p. 3.

System Structure Diagram (SSD) A System Structure Diagram graphically represents the static structure (entities and relationships) of a system. In our methodology it is used to represent the services and the relationships (directed logical channels) between them, p. 26.

U

usage behavior The usage behavior of a system is the totality of functions/services that a user can call, p. 3.

user In this work, the term user refers to both human users and other (technical) systems with which a system communicates, p. 53.

V

vertical (service) relationship In our approach we distinguish between vertical and →horizontal (service) relationships. Vertical relationships describe the structure of (comprehensive) services, i.e. they describe out of which subservices a service is comprised, p. 62.

Appendix A

Embedding of the approach into a theoretical framework

So far, we have introduced and treated the system model which underlies our approach (see Section 4.1.1, *Underlying system model*) informally. The reason for this is that informal descriptions are usually easier to understand and more pragmatic to use. However, informality also has disadvantages: in order to precisely describe an approach a formal, theoretical basis is needed.

We now embed our concepts into a theoretical framework and compare it in detail with another service-oriented specification style, namely with the predicate-based specification of services. In Section A.1 (*JANUS - A theory for service-orientation*) the formal model which implicitly lies behind our approach - the FOCUS/JANUS theory - is described. In Section A.2 (*Relation of our approach to the JANUS theory*) we relate our concepts to this theoretical framework. We compare our approach to a predicate-based, service-oriented approach in Section A.3 (*Comparison to another approach*).

The contents presented in this section also are concerned with comparing our approach to other approaches and thus could be placed in the related work section (see Chapter 6, *Related Work*). However, as the FOCUS/JANUS theory provides the formal framework for our approach we present it separately and in more detail in this section. Due to the fact, that the predicate-based specification approach (see Section A.3.1, *Predicate-based specification of services*) also fits into the formal FOCUS/JANUS framework, we also present it here.

A.1. JANUS - A theory for service-orientation

In this section, we describe the formal model behind our approach: the JANUS approach [Broy, 2005]. The main goals of JANUS are to give a formal model for services, layers, and layered architectures. Furthermore, a theory for relating, composing, and refining services, layers, and layered architectures is aimed at. Advanced goals are specification and verification techniques, a methodology for designing services and respective architectures, and design patterns for services, layers, and layered architectures.

The JANUS approach is based on the FOCUS system model [Broy and Stolen, 2001]. The aim of FOCUS is to first specify systems as families of components (and their interfaces) and then to put the components together forming the architecture. The verification of the component specification can be achieved by the interface specifications of the components and the composition verification rules.

In the following section (see Section A.1.1, *The FOCUS Approach*) we describe the FOCUS approach. In Section A.1.2 (*The JANUS approach*) we give an introduction to the JANUS approach.

A.1.1. The FOCUS Approach

In the FOCUS model, a system is composed of a number of components which encapsulate behavior (principle of modularity). The notion of system is relative, i. e. components can be composed to components again: "A system is a component is a system" [Broy and Stolen, 2001].

Interfaces exist between components and between components and the environment. The communication between components and with the environment takes place via directed communication channels. A mapping of message streams to channels represents the exchange of typed messages over these channels assuming a system wide global clock. The behavior of such systems is expressed as equations or relations relating a number of (possibly infinite) input sequences to a respective set of output sequences. More precisely, the communication is expressed in terms of relations on streams; streams represent histories of communications of data messages in a time frame. Multiple possible outputs for a certain input express non-determinism. Furthermore, the channels are divided into disjoint input and output channels for a component, respectively. Causality of such a system is enforced by requiring that every computed output is only dependent on inputs which have been received before the output happens.

In the following we formally describe the basic concepts of the FOCUS approach. For more information please refer to [Broy and Stolen, 2001].

(In-) Finite (Non-) timed streams

As mentioned above, streams represent histories of communication of data massages in a given time frame. Let M be the universe of messages; by M^* (M^∞) the set of finite (infinite) sequences of elements of M is denoted. It can be represented by a mapping $N \to M$ where N is the set of natural numbers without 0. M^* (M^∞) is called a finite (an infinite) non-timed stream. For a given set M, a *timed stream* is defined by

$$s : N \to M^*$$

whereas $s(t)$ stands for the messages communicated at time t in the stream s.

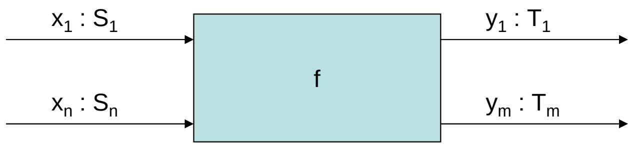

Figure A.1.: Graphical representation of a FOCUS component and its typed input and output channels

Input/Output channels and channel histories

Channels are used as identifiers for streams. They are divided into disjoint sets of *input channels I* and *output channels O*. $Type(c)$ denotes the data type of messages sent over the channel c. To that end, the function $Type : C \to TYPE$ is given, with $TYPE$ being a set of types which are carrier sets of data elements.

Let C be a set of typed channels; a *channel history H(C)* is a mapping

$$x : C \to (N \to M^*)$$

such that $x.c$ is a stream of type $Type(c)$ for each $c \in C$.

Furthermore, the following notations are introduced on streams: $s.k$ is the *k-th sequence* in the stream s. $s{\downarrow}k$ is the *prefix of length k* ($k \in N$) of the timed stream s.

Components

The behavior of components is described by their black box behavior, i. e. their interfaces. The interfaces provide a syntactic and semantic notion. The syntactic interface associates a type for the component whereas the semantic interface describes the observable behavior. Let I and O be sets of typed input and output channels, respectively.

$$(I \blacktriangleright O)$$

denotes the *syntactic interface* of a component. It defines the types of messages that can be exchanged by a component. The *semantic interface* with the syntactic interface $(I \blacktriangleright O)$ is represented by

$$F : H(I) \to \wp(H(O))$$

that fulfills the timing property for all its input histories.

Let $x, z \in H(I)$, $y \in H(O)$, and $t \in N$. The *timing property* is specified as follows:

$$x{\downarrow}t = z{\downarrow}t \Rightarrow \{y{\downarrow}(t+1) : y \in F(x)\} = \{y{\downarrow}(t+1) : y \in F(z)\}.$$

This means that the set of possible output histories for F for the first $t{+}1$ intervals only depends on the inputs of the first t time intervals. Or in other words: The processing of messages takes at least one time interval. Functions that fulfill this timing property are called time-guarded or strictly causal. The application of a strictly causal function leads to either an empty output for all input histories, or a non-empty output for all input histories. In the first case, we call the function paradoxical, in the second case we call it total.

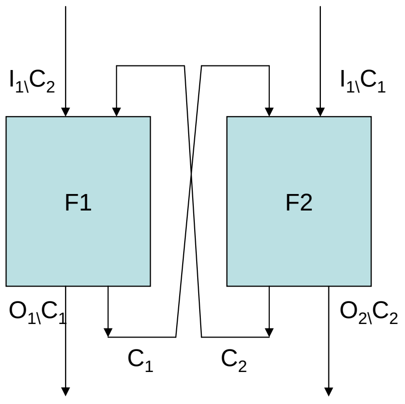

Figure A.2.: Composition of components in the FOCUS theory

A component is a *total behavior*. Figure A.1 gives a graphical representation of a FOCUS component.

Composition of components

The composition is done by parallel *composition with feedback* [BS01]. Figure A.2 schematically shows how two components $F1$ and $F2$ can be composed by the introduction of feedback channels $C_1 \subseteq O_1 \cap I_2$ and $C_2 \subseteq O_2 \cap I_1$.

Formally: Let F be the compositon of the components $F1$ and $F2$ having the syntactic interfaces $(I_1 \blacktriangleright O_1)$ and $(I_2 \blacktriangleright O_2)$, respectively. The syntactic interface of F is denoted by $(I \blacktriangleright O)$ with $I = (I_1 \backslash O_2) \cup (I_2 \backslash O_1)$ and $O = (O_1 \backslash C_1) \cup (O_2 \backslash C_2)$. Let $z \in H(I_1 \cup O_1 \cup I_2 \cup O_2)$ and $x \in H(I)$. The semantic interface of F is defined by

$$F.x = \{z|O : x = z|I \wedge z|O_1 \in F_1(z|I_1) \wedge z|O_2 \in F_2(z|I_2)\}$$

Specification styles

FOCUS suggests different styles for the specification of the system behavior [Broy and Stolen, 2001]:

Equational specifications are specifications which define the behavior in terms of recursively defined stream processing functions.

Assumption/Guarantee specifications (abbr.: A/G specifications) structure in the specification into an assumption and a guarantee part. The assumption describes the properties expected of the input which is legal for the specification. The guarantee part describes the black box behavior of the component that the system must fulfill in case the assumption is fulfilled.

Tables and diagrams (like State Transition Tables and State Transition Diagrams) are also provided as specification styles.

A.1.2. The JANUS approach

In this section, we describe the enhancement of the FOCUS approach for service-orientation: the JANUS approach. We again only summarize the main concepts which we need for our work. For more information please refer to [Broy, 2005].

Component interfaces provide functionality. The idea is to describe for each piece of functionality (method/message) under which conditions (precondition) the functionality may be invoked (message may be sent or received) and which effects this has (postcondition). This leads to the notion of a service.

Services, service range, and service domain

A service is a *partial behavior* as opposed to components which are total behaviors. Partial means that a service is defined only for a subset of input histories. This subset is called the service domain. A service is a set of interaction patterns. Formally, a service with the syntactic interface $(I \blacktriangleright O)$ is given by a function

$$F : H(I) \rightarrow \wp(H(O))$$

that fulfills the timing property only for the input histories with nonempty output set. Let $x, z \in H(I)$, $y \in H(O)$, and $t \in N$. Then:

$$F.x \neq \varnothing \neq F.z \wedge x{\downarrow}t = z{\downarrow}t \Rightarrow \{y{\downarrow}t+1 : y \in F(x)\} = \{y{\downarrow}t+1 : y \in F(z)\}$$

The set

$$Dom(F) = \{x : F.x \neq \varnothing\}$$

is the so-called *service domain*. It characterizes those input streams for which F is defined.

$$Ran(F) = \{y \in F.x : x \in Dom(F)\}$$

is the so-called *service range* and defines the output for all valid inputs. A service gives a partial view onto the behavior of a component.

For services, the same specification styles (see above) as for components can be used.

Interface subtyping and the projection of a behavior

Before describing various service relationships, we first explain some basic concepts.

Given a typed channel set $C1$ and a typed channel set $C2$. $C1$ is called a *subtype* of $C2$ if the following formula holds:

$$(c, T1) \in C1 \Rightarrow \exists T2 \in TYPE : (c, T2) \in C2 \wedge Type(T1) \subseteq Type(T2)$$

We then write $C1\ subtype\ C2$.

$(I_1 \blacktriangleright O_1)\ subtype\ (I \blacktriangleright O)$ is an abbreviation for $(I_1\ subtype\ I_2) \wedge (O_1\ subtype\ O_2)$.

Let $(I_1 \blacktriangleright O_1)$ and $(I \blacktriangleright O)$ be two syntactic interfaces with $(I_1 \blacktriangleright O_1)\ subtype\ (I \blacktriangleright O)$, the projection for a behavior F with the syntactic interface $(I \blacktriangleright O)$ to the syntactic interface $(I_1 \blacktriangleright O_1)$ is defined by the following equation:

$$\forall x \in H(I_1) : F\dagger(I_1 \blacktriangleright O_1).x = \{y|O_1 : \exists x' \in H(I) : x = x'|I_1 \wedge y \in F.x'\}$$

where $x|I$ is the stream x projected to the channels and types of I.

Service relationships

Based on the concepts above, [Broy, 2007] introduces vertical service relationships. Horizontal service relationships are not further discussed in [Broy, 2007] (but mentioned to be important).

We now explain service refinement and sub service relationships. Let $F1$ have the syntactic interface (I_1►O_1) and $F2$ have the syntactic interface (I_2►O_2), respectively, with (I_1►O_1) $subtype$ (I_2►O_2). $F2$ is a *service refinement* of $F1$ if for all input histories $x \in H(I_1)$:

$$F\dagger(I_1\blacktriangleright O_1).x \subseteq F_1.x$$

Thus, it is permitted to enlarge the number of channels and their types. As the paradoxical services is always a service refinement, this notion of refinement is too liberal. Consequently, [Broy, 2007] furthermore introduces the notion of a sub service relation.

$F1$ is a *sub service* of $F2$, if $F2$ is a service refinement of $F1$ and the following formula holds:

$$Dom(F1) \subseteq Dom(F2\dagger(I_1\blacktriangleright O_1))$$

In cases in which the sub service relationship does not hold, there may hold a restricted version, if there exists a set $R \subseteq Dom(F2)$ of input histories such that $F1$ $subservice$ $F2|R$. This means that $F1$ is a sub service for the input histories of a subset of the domain, namely of R. This leads to the following definition. Let $F1$ and $F2$ be as above. $F1$ is called a *restricted sub service* of $F2$, if there exists a set R of input histories with $R \subseteq Dom(F2)$ such that $F1$ $subservice$ $F2|R$ holds.

[Broy, 2007] suggests to use trees to graphically display services according to the (restricted) sub service relationship.

A.2. Relation of our approach to the JANUS theory

In this section, we relate our approach to the formal framework of JANUS. As our main concepts (underlying system model, services, and service relationships) are based on this model (see Section 4.1.1, *Underlying system model* and 5.4.3, *Formal specification of modular services*) we will see that the mapping works well.

Note that as JANUS does not provide methodological support, we only relate the concepts of the two approaches.

A.2.1. Relation of the underlying system models

As mentioned in the previous paragraph, our underlying system model can be easily mapped to the FOCUS system model: In our approach, the stimuli that go into the system are the so-called input actions. They are simply the input messages of the

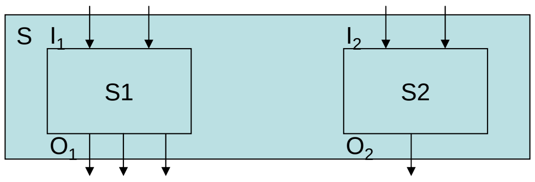

Figure A.3.: Combination of independent services

FOCUS world. Analogously, the responses of the system which are our output actions can be seen as the output messages of FOCUS. The sum of the input and output messages thus is the universe of messages M.

The behavior in both approaches is described by relating inputs to outputs. In our case, the automata describing the service behaviors, turn sequences of inputs (input histories in FOCUS) step by step to sequences of outputs (output histories in FOCUS). A service thus is a FOCUS service $S : H(I) \rightarrow \wp(H(O))$ relating an input history to an output history of logical actions (messages).

We also - like FOCUS / JANUS - assume a global clock that structures time (time synchronous system). Furthermore, we also specify strongly causal behavior as the output in time interval $t+1$ depends on the input until time interval t. Consequently our notion of service fulfills the requirement that a service has to fulfill the timing property.

The matching system models of our approach and the FOCUS approach in fact are the basis for having chosen AutoFOCUS State Transition Diagrams (see Section 3.2, *State Transition Diagrams (STDs)*) as notational technique.

A.2.2. Relation of service-oriented concepts

As already mentioned in Section 5.4.3 (*Formal specification of modular services*), we make use of the JANUS service definition and define a service as a partial behavior. Therefore, we obtain the same consequences, like a relative notion of services and the possibility of underspecification and the specification of non-determinism.

[Broy and Stolen, 2001] describes different specification styles for the specification of behavior (see Section A.1.1, *The FOCUS Approach*). In our methodology, we make use of the specification style "tables and diagrams" as we use State Transition Diagrams which is an operational specification technique.

As far as service relationships are concerned, [Broy, 2007] introduces (above others) the sub service relationship and the restricted sub service relationship which we also make use of (see Section 4.6, *Identification of service relationships*). In [Broy, 2007] however, a super service only has to "contain" the sub services. It is theoretically possible that it also offers arbitrary additional behavior. In our approach, the super service is exactly the combination of the sub services plus the realization of the service relationships.

As far as the combination of services is concerned, the ideas of [Broy, 2007] and of our work also match. In [Broy, 2007] the combination of two independent services, i. e.

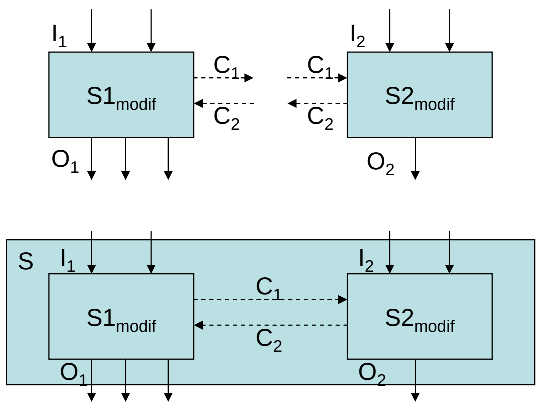

Figure A.4.: Combination of dependent services

services with disjoint input and output interfaces, is done as depicted in Figure A.3. The super service S being comprised of the behaviors $S1$ and $S2$ can be obtained by simply putting the sub services next to each other.

In the more common case that two services exhibit dependencies between each other, the combination is more intricate. The two modular service specifications $S1$ and $S2$ first have to be refined accordingly to handle the service influence. Figure A.4 shows how this is done. In the upper part of the figure, the services are refined by the introduction of additional channels. On these channels, the logical actions which are needed to realize the service relationship are sent. We obtain the modified service behaviors $S1_{modif}$ and $S2_{modif}$ having the syntactic interfaces

$$((I_1 \cup C_2) \blacktriangleright (O_1 \cup C_1))$$

and

$$((I_2 \cup C_1) \blacktriangleright (O_2 \cup C_2))$$

respectively. The super service S is then obtained by composing the modified sub services. The modification of the services depends on the service relationships between the services.

Formally, it is required that the modified behaviors $S1_{modif}$ and $S2_{modif}$ contain the modular service specifications $S1$ and $S2$ as restricted sub services, i. e. that the following formulae hold: $S1\ restrictedsubservice\ S1_{modif}$ and $S2\ restrictedsubservice\ S2_{modif}$. In those cases that no service interaction takes place, the service specifications behave according to their modular specification. The input histories are then in the subset R for which the service is a sub service of the super service (see above).

In our methodology, the introduction of additional services (e. g. conflict solving services) might also be necessary. However it just matches to the introduction of further services and channels.

A.3. Comparison to a predicate-based service-oriented approach

In the previous section we embedded the concepts of our approach in a theoretical framework. In this section we describe another service-oriented approach which also fits nicely into the JANUS framework. Although this other approach makes use of the same understanding of services, it uses a different specification style, namely property-based specification. We will also explain their concepts with help of the FOCUS and the JANUS theory and finally compare it to our approach and give advantages and disadvantages of both approaches.

A.3.1. Predicate-based specification of services

The approach described in this section is currently elaborated in the VEIA[1] project. The aim of VEIA is to develop concepts for the distributed development and integration of automotive software systems. They also make use of services to formalize the system functionality. Modular service specifications are combined to obtain the overall system behavior step by stop. Based on a service-oriented system functionality, the model is enriched by product line aspects like variability issues [Hartmann and Harhurin, 2008].

The VEIA approach also defines services as pieces of partial behavior. Each service has a syntactic interface determining which messages can be received and sent by the service. The service behavior is a partial mapping relating input streams to output streams. The single services are specified in an assumption/guarantee manner. The assumption of a service is a predicate on the input streams and indicates which streams are a valid input for the service. The guarantee is a predicate on both input and output streams and defines the behavior of the service in case a valid input has been made to the service.

Formally (see [Hartmann and Harhurin, 2008]): Let $(I \blacktriangleright O)$ be the syntactic interface of a service S.[2] The assumption and the guarantee of a service have the following signatures, respectively:

$$\begin{aligned} A_S &: H(I) \to \mathbb{B} \\ G_S &: H(I) \times H(O) \to \mathbb{B} \end{aligned}$$

A service is consequently defined as:

$$\begin{aligned} S &: H(I) \to \wp(H(O)) \\ S(x) &\equiv \{y | A_S(x) \wedge G_S(x, y)\}. \end{aligned}$$

Consider, as a small example, the power window services (including a child protection service) as depicted in Figure A.5. The power window services allow the user

[1]Verteilte Entwicklung und Integration von Automotive Produktlinien, BMBF project, grant number 01ISF15A.

[2]In [Hartmann and Harhurin, 2008] the interface is defined by the typed input and output ports of a service instead of its typed input and output channels. Therefore, I and O are defined over ports. However this difference does not matter for our comparison.

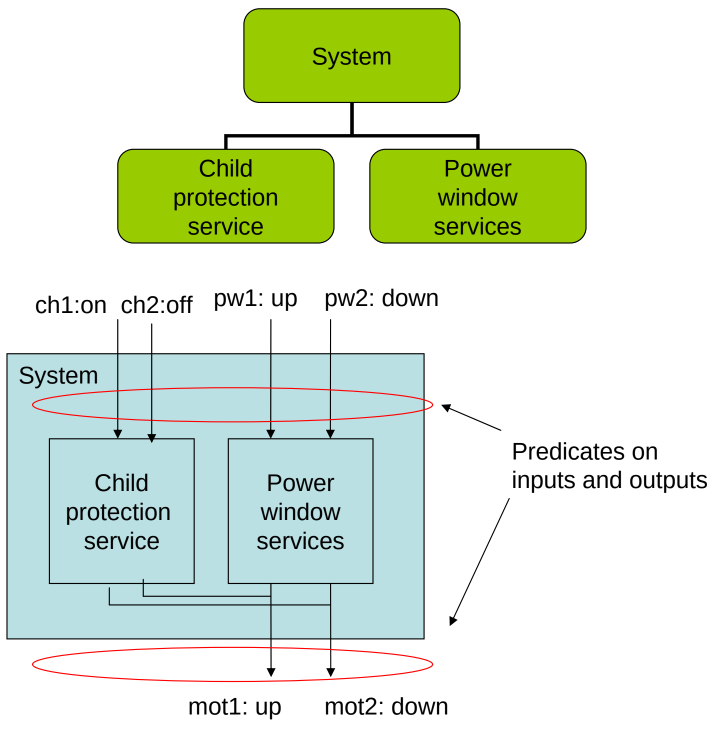

Figure A.5.: Predicate-based specification of services

to move the windows up and down. If the child protection service is enabled, the windows can not be moved, i. e. the user's wish is ignored. The assumptions and guarantees of the power window services (PWS) is:

$$\begin{aligned} A_{PWS}(x) &\equiv \forall t \in \mathbb{N} : (x[pw1](t) \in \{up, \epsilon\} \wedge x[pw2](t) \in \{down, \epsilon\}) \\ G_{PWS}(x,y) &\equiv \forall t \in \mathbb{N} : (x[pw1](t) = up \Rightarrow y[mot1](t) = up \wedge \\ &\qquad x[pw2](t) = down \Rightarrow y[mot2](t) = down) \end{aligned}$$

with $x \in H(I_{PWS} \cup I_{CPS})$ and $y \in H(O_{PWS} \cup O_{CPS})$ and $x[pw1](t)$ depicting the messages sent on channel $pw1$ within the input stream x in time interval t. The assumption demands that the input on the channel $pw1$ ($pw2$) is either up ($down$) or nothing (ϵ). The guarantee assures that if the driver wants to move the window up (down), the respective command is sent on the output channels.

The child protection service (CPS) has the following assumption and guarantee:

$$\begin{aligned} A_{CPS}(x) &\equiv \forall t \in \mathbb{N} : (x[ch1](t) \in \{on, \epsilon\} \wedge x[ch2](t) \in \{off, \epsilon\}) \\ G_{CPS}(x,y) &\equiv \forall t \in \mathbb{N} : (x[ch1](t) = on \Rightarrow y[mot1](t) = \epsilon \wedge y[mot2](t) = \epsilon \end{aligned}$$

The inputs for the child protection service can either be on, off, or nothing (ϵ). If the child protection service is turned on, no output is allowed to be sent on the output channels $mot1$ and $mot2$.

The modular service specifications (i. e. predicates) are then combined by logical conjunction.

$$System\_Specification = S_{CPS} \wedge S_{PWS}$$

During this combination, conflicts may occur being the consequence of feature interaction. For example, the child protection service demands that no output is sent on the channels $mot1$ and $mot2$ in case the child protection is turned on. However, the power window services demand the controlling of the respective motor in case the user wants to move the window. Thus, different demanded outputs on the same output channel within the same time interval cause a conflict.

In order to solve conflicts, additional predicates (conflict solving predicates) are introduced. They can be seen as the formalization of service relationships. In the example above, the predicate S_{PWS} is then loosened and replaced by the new conflict solving predicate CPS_{prio}:

$$\begin{aligned} CPS_{prio} \equiv &\forall t \in \mathbb{N} : (CPS[y][mot1](t) \neq PWS[y][mot1](t) \Rightarrow CPS[y][mot1](t) \wedge \\ &CPS[y][mot2](t) \neq PWS[y][mot2](t) \Rightarrow CPS[y][mot2](t)) \end{aligned}$$

where $CPS[y][mot1](t)$ denotes the messages sent on channel $mot1$ within the stream y during time interval t as specified by the service CPS.

The overall system specification finally is given by conjunction of all service predicates and conflict solving functions.

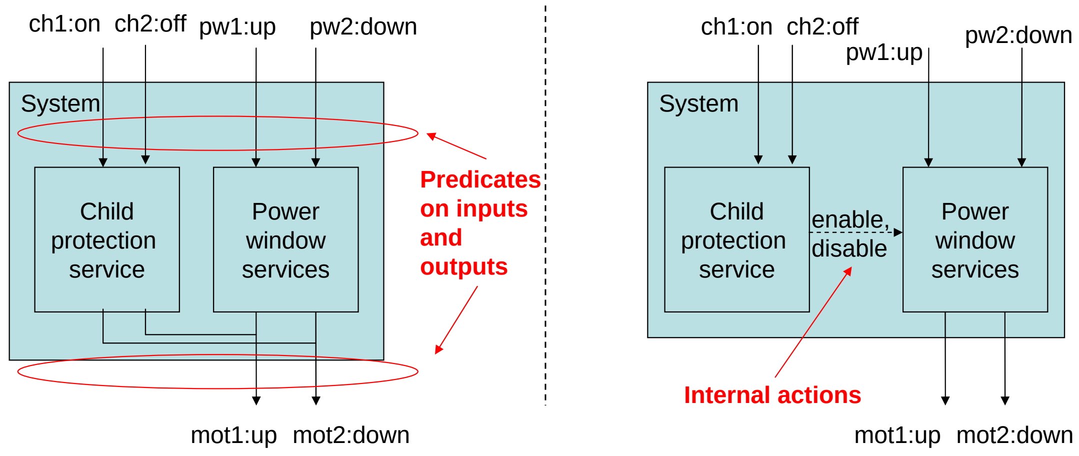

Figure A.6.: Comparison of both specification techniques (l.h.s: VEIA approach; r.h.s: our approach)

A.3.2. Comparison of both approaches

Although the VEIA approach and the approach introduced in this thesis share some of the same ideas, they also differ in many respects. In this section we compare the approaches.

Commonalities Both approaches have the same underlying system model. Furthermore, they aim at the formalization of functional requirements and thus at bridging the gap between the requirements engineering and the design phase. Services - which are pieces of partial behavior - are the main entities of which a system specification is comprised. The understanding that modular service specifications have to be adapted to handle service relationships is also common to both approaches. In the VEIA approach this is achieved by substituting service predicates by special conflict solving predicates. In our methodology, feature interaction is handled by the introduction of dedicated conflict solving services.

Differences VEIA elaborates concepts for the specification of service-oriented systems (and enhances them with concepts for product line development). The aim of our work is to provide an overall methodology which is based on service-oriented concepts.

The main differences between the approaches can be described by looking at Figure A.6. The figure confronts the different specifications of the child protections service and the power window services. On the left hand side, the VEIA specification can be seen. The system specification as proposed in this thesis is depicted on the right hand side. Both approaches modularly specify the child protection service and the power window services. VEIA makes use of predicates and thus uses a predicate-based specification style whereas we make use of automata to specify the behavior (see Section 5.4.3, *Formal specification of modular services*) and consequently make use of an operational specification style. The combination of the sub services is done differently, too: VEIA uses additional predicates on inputs and outputs in order to handle service relationships, i. e. feature interaction. In our methodology we make use of internal actions to control the influences between services. The modular service specifications are adapted accordingly.

The VEIA project does not provide a standard set of predicates for the handling of service relationships. For each service relationship a new predicate has to be defined. Future work of the VEIA project will deal with the introduction of classes of predicates. For example, a class for the specification of priorities as needed in the example above. Predicates of these classes would then have the same structure/pattern.

Advantages and disadvantages The VEIA approach is more abstract. It abstracts from internal communication, internal control and data states. This information is represented in the predicates. Consequently, a stricter black box view is kept. However, modularity is given up as these predicates refer to several services. In our approach, the power window services do only have to be able to handle a DISABLE and an ENABLE command, but do not need the information why this command is sent. Furthermore, we provide a set of standard relationships which can be used for the combination of the services. In the VEIA project a new predicate has to be introduced in each case (see previous paragraph).

In Section 4.8 (*Combination of services on basis of the service relationships*) we showed how the modular formal service specifications can be combined on basis of the service relationships between them. Thus, we exhibit a stronger compositionality of services.

The result of the VEIA project is a conjunction of logical formulae. These can be verified by theorem proving. In contrast, the result of our approach is an executable model of the system functionality which can be simulated and validated.

As VEIA is more abstract than our approach, its approach faces a conceptual gap between the predicates and the logical architecture which is comprised of logical components and internal communication channels, etc. The transition to a logical architecture having an operational semantics is more difficult. The question arises how to perform the step from logical formulae (the result of the VEIA approach) to a logical architecture. The opposite holds for our approach: Here, the formal model of the requirements is more concrete from the beginning as it uses logical entities and communication channels between them. However, the requirements model consequently already contains information how to realize the functional requirements. We think that due to this reason our approach is more pragmatic for industry.

The question which specification style is better - predicate-based or operational - is a question of taste and can not be answered in general.

A.3.3. Conclusion

In the previous subsection, we compared our approach with the VEIA approach. To put it in a nutshell, the VEIA approach can be situated on a more abstract level. The advantages and disadvantages of both approaches are basically a consequence of these different levels of abstraction.

As VEIA mainly uses the same concepts but on a more abstract level, the idea arises

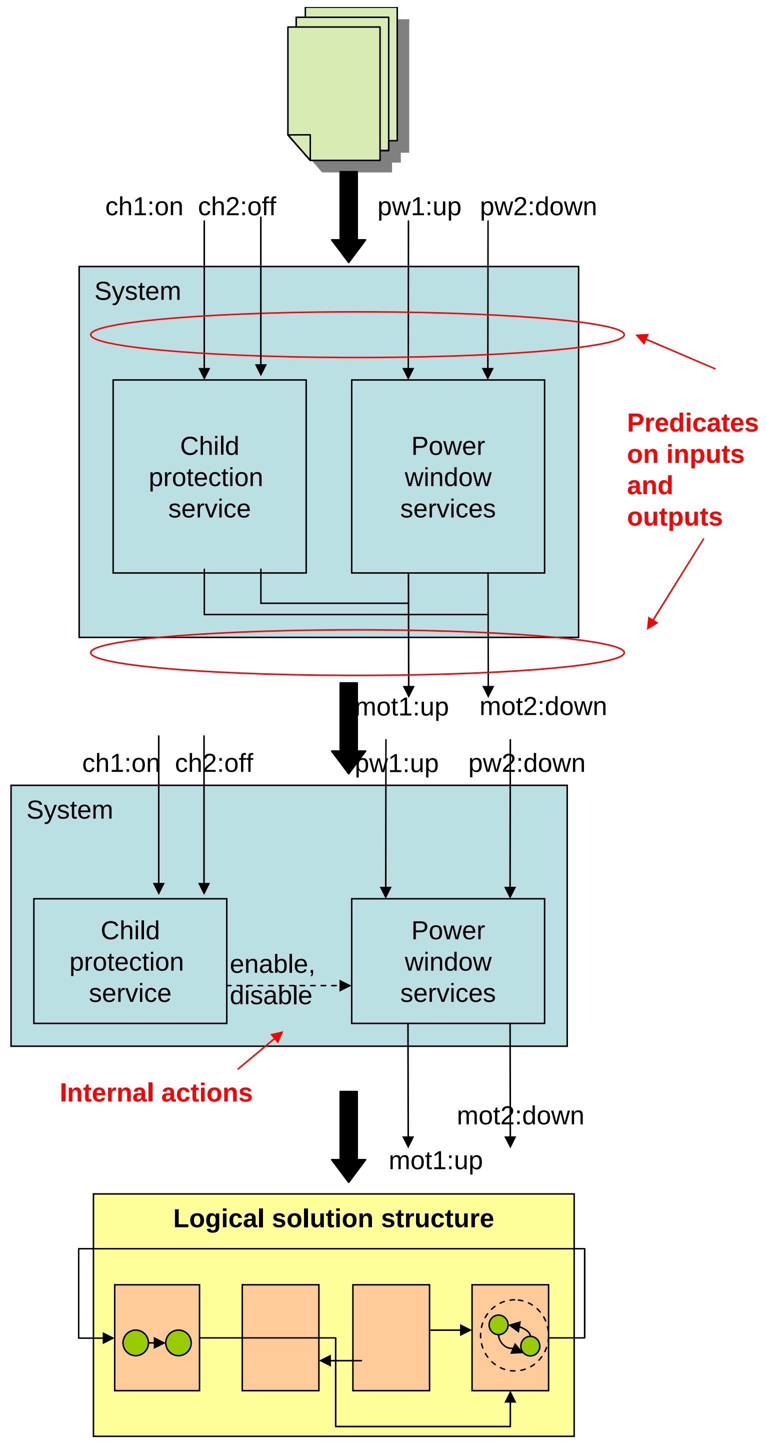

Figure A.7.: Integrated requirements engineering phase with both approaches

if VEIA and our presented approach can be combined. Figure A.7 shows this idea graphically. The textual requirements would then be first specified very abstractly by predicates and then mapped to a first operational logical architecture.

First discussions with the VEIA project team lead to the assumption that the consecutive ordering of the two approaches is possible. However, the question is how pragmatic such an integrated model would be or if it would be an overkill. Respective cost benefit analyses would have to be carried out.