Model-based Requirements Engineering for Multifunctional Systems

Andreas Vogelsang

Technische Universität München

Institut für Informatik
der Technischen Universität München

Model-based Requirements Engineering for Multifunctional Systems

Andreas Vogelsang

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. Florian Matthes
Prüfer der Dissertation:		
	1.	Univ.-Prof. Dr. Dr. h.c. Manfred Broy
	2.	Univ.-Prof. Dr. Sven Apel, Universität Passau

Die Dissertation wurde am 15.12.2014 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 20.03.2015 angenommen.

Abstract

Many technical systems that were initially designed for serving one purpose now offer a number of functions integrated side-by-side into one application or device. A typical example for such a multifunctional system is the software system of an automobile, which provides functions ranging from multimedia applications over driver assistance functions to diagnosis functions. Other examples are telecommunication or avionic systems.

Specifications of multifunctional systems can grow large and therefore need to be structured to support its development and evolution. Structuring large specifications into functions in a compositional way is highly desirable since functionality can be specified independently from each other. Unfortunately, different functions within a specification of a multifunctional system are often interconnected or influence each other, which makes it hard to describe them independently in a compositional way. Formal and model-based specification techniques have been proposed by the research community to facilitate the specification of multifunctional systems. However, these techniques are rarely applied in industry. In our experience, the reason for this is a missing integration of the proposed formal specification techniques for multifunctional systems into a comprehensive requirements engineering (RE) methodology that describes the stepwise creation and analysis of specifications based on the formal specification technique.

This thesis provides supporting evidence and solutions for the stated problem. First, we present an empirical analysis of function dependencies in productive multifunctional systems. The analysis reveals that functions are highly interdependent and current engineering approaches address this issue only insufficiently. Furthermore, the study reveals that developers are not aware of these dependencies in most cases.

Second, we present the integration of a formal specification model for multifunctional systems into a comprehensive requirements engineering methodology. In the formal specification model, a multifunctional system consists of *functions*, which can have dependencies that are described by *modes*. We integrate these formal concepts into a requirements engineering methodology by providing an artifact model that relates the concepts to model types that we use to describe them. The semantic relations between the concepts are reflected and expressed by the model types. Furthermore, we describe the role of the resulting artifacts in a development process. Our main contribution is the definition of this methodology that allows a stepwise development of a formal and model-based specification for multifunctional systems. We instantiate the methodology (1) in a scenario-driven RE context and (2) in a property-driven RE context and demonstrate its effectiveness through two case studies, where the methodology increased artifact consistency and detection of specification flaws.

Finally, we highlight two central artifacts, namely the *function documentation* and the *mode model*, and provide detailed instructions on how to elicit and structure these artifacts. The proposed structure for function documentation is empirically grounded by an exploratory qualitative study in the context of automotive systems. The elicitation of a mode model is demonstrated through an industrial case study, showing its feasibility and discussing different elicitation approaches.

Acknowledgements

This thesis would not exist without the tremendous support of a number of people whom I am deeply thankful. I want to expresses my gratitude for Prof. Dr. Dr. h.c. Manfred Broy who gave me the opportunity to work as a researcher in his stimulating, supportive, and exceptional research group. His guidance and advice always broadened my horizon and his commitment to create opportunities in many ways is outstanding.

I want to thank Prof. Dr. Sven Apel for co-supervising my thesis. His comments and insights, especially in the field of feature-oriented software development, helped me a lot.

Important parts of this thesis could not have been realized without the support of industrial partners. My thanks go to Stefan Teuchert, Christian Winkler, and Jean-François Girard, who are with MAN Truck & Bus AG, to Jochen Quante and Andreas Thums, who are with Robert Bosch GmbH, to Steffen Fuhrmann who is with BMW Group, and to Ralf Pinger, who is with Siemens AG.

I count myself lucky to work with colleagues who already were or became friends. I am especially thankful to the members of the "Zwiebel" group: Veronika Bauer, Jonas Eckhardt, Sebastian Eder, Henning Femmer, Benedikt Hauptmann, Maximilian Junker, and Daniela Steidl. Additionally, I want to thank Mario Gleirscher, Georg Hackenberg, Lars Heinemann, Thomas Kofler, Daniel Méndez Fernández, Silke Müller, Jakob Mund, and Sabine Teufl for their assistance and contributions to a warm and supportive working environment.

During my time as a research assistant, I learnt a lot about research in general, business, personal development, and professional work. I want to thank Wolfgang Böhm, Martin Feilkas, Elmar Jürgens, Andreas Koller, and Bernhard Schätz for sharing their professional and personal experiences with me.

Last but not least, I want to thank all the people around me who supported me in any possible way. I am deeply grateful to my family, my friends and, above all, my wife Lisa.

"The structure of system functionality is more like colour separation than it is like an assembly of parts."

– Michael Jackson

Contents

Chapter 1

Introduction

The topic of this thesis is a model-based requirements engineering and specification methodology for multifunctional systems. In this chapter, we introduce and motivate this topic by characterizing multifunctional systems (Section 1.1) and stating the problems related to the specification of their functional requirements (Section 1.2). In Section 1.3, we summarize the major contributions of this thesis. In Section 1.4, we briefly present our basic approach to model-based specifications of multifunctional systems, before we finally provide an outline of this thesis in Section 1.5.

1.1 Context: Multifunctional Systems

Systems that offer a variety of different functions to their environment are called multifunctional systems [Broy, 2010b]. A function describes the intent to use a system for a specific purpose. Many technical systems that were initially designed for serving one purpose now have a number of functions integrated side-by-side into one application or device serving a number of purposes. A typical example for a multifunctional system is the software system of an automobile, which provides functions ranging from multimedia applications, driver assistance functions, to diagnosis functions. Other examples are telecommunication or avionic systems. Within these systems, the desired functions are realized by a set of (implementation) components arranged in an architecture.

Different functions of a system serve different purposes and can be described and realized independently to some extent. Therefore, function-oriented development, where functions are mirrored by implementation components that are developed independently by different teams and even by different departments, is prevalent in companies that build multifunctional systems [Broy et al., 2007a].

However, the functions of a multifunctional system can have subtle dependencies and may affect each other in certain situations. For example in a car, the central locking function and the crash sensing function behave independently to a large extent, however, in case of an accident, the crash sensing function forces the central locking function to unlock all doors of the car. We call this a *function dependency*. Function dependencies can be considered as one form of *feature interaction* [Calder et al., 2003; Zave, 2001]: *"A feature interaction is some way in which a feature or features modify or influence another feature in defining overall system behavior.*[1]*"* [Zave, 1999]. Feature interaction may result in desired or undesired behavior. This thesis focuses on the specification of multifunctional systems and not on their implementation. Therefore, the term function dependency, as used in this thesis, refers to interaction between functions used to specify the desired behavior of a multifunctional system. Undesired feature interaction often arises from function *implementations* that are integrated into one system and then interact in an undesired way. This is another interpretation of the term feature interaction, which we do not address with the term function dependency. Moreover, the implementation of a system in terms of components may follow a completely different structure than induced by the functions (cf. *tyranny of the dominant decomposition* [Tarr et al., 1999]). This is especially true if the implementation contains shared components that contribute to the realization of several functions (e.g., a sensor fusion component).

Specifications of multifunctional systems can grow large and therefore need to be structured to support their development and evolution. Structuring large specifications into functions in a compositional way is highly desirable because the system specification is then given by the composition of modular function specifications that encapsulate behavior serving one purpose. The mentioned function dependencies need to be considered in a compositional specification. The characteristics of multifunctional systems challenge their specification due to the following reasons:

- Functions of a multifunctional system cannot be specified in isolation. They exhibit subtle dependencies that need to be considered within their specification.
- Without the decomposition into functions, specifications of multifunctional systems become over-complex, hard to maintain, and impossible to validate modularly.
- The implementation of a multifunctional system may follow a completely different structuring paradigm than a structuring according to functions.

As a consequence, specifications for multifunctional systems are often structured into functions. However, these functions are generally developed in isolation and their behavior is often specified on an architectural/implementation level (e.g., by linking function names to code artifacts). This leads to integration errors and unwanted behavior due to function dependencies, and blurs the distinction between functional and architectural dependencies.

1.2 Problem Statement

Due to these characteristics, different groups in the software and systems engineering community have recognized the importance of function-oriented specification and development, and investigated its benefits and limitations. In the corresponding work, dependencies between functions have been considered as a major factor leading to integration failures in multifunctional systems [Benz, 2010; Cataldo and

[1] In this context, the term *feature* is a synonym for the term *function*.

Herbsleb, 2011]. In an automotive context, Broy [2006] states, *"So far, the understanding of these interactions between the different functions in the car is insufficient."*

Formal and model-based specification techniques have been proposed by the research community to facilitate the specification of multifunctional systems (e.g., Broy [2010b]; Heitmeyer et al. [1997]; Jackson and Zave [1998]; Schätz [2008]). Most of them aim at a modular specification for multifunctional systems. The system is broken down into functions, which are described separately. Dependencies between functions are modeled by different extensions to their interfaces.

Despite the stated challenges resulting from function dependencies and the proposed specification techniques to address these, function specifications in industry currently do not consider function dependencies. In our experience, the reason for this is a missing integration of the proposed formal specification techniques for multifunctional systems into a comprehensive requirements engineering methodology.

Formal and model-based specification techniques for multifunctional systems need to be integrated into a comprehensive requirements engineering methodology to be applicable in industry. This includes the extension and adaption of RE artifacts and analysis techniques applied to them. However, for the existing specification techniques such an integration is missing. For the few techniques that come with tool support, it is not clear how the tools can be integrated into an existing RE process, and how the existing RE artifacts relate to those added by the specification technique. Approaches that abstain from extending or adapting existing artifacts, such as UML diagrams, lack of precision and are not expressive enough to describe the subtleties of function dependencies. Thus, an integration of formal specification techniques for multifunctional systems into a comprehensive requirements engineering methodology is required, including the description of artifact types, their relations, associated analysis procedures, and their role in a development process.

> **Problem Statement:**
> We need a comprehensive requirements engineering methodology for multifunctional systems that integrates and supports the specification of function dependencies.

1.3 Contributions of this Thesis

In this thesis, we provide supporting evidence and solutions for the stated problem.

Significance of Function Dependencies in Multifunctional Systems We present an empirical study on two productive automotive systems with the goal to assess the extent and characteristics of function dependencies as well as the developer's awareness of those. In the examined systems, the function dependencies were not part of the function specifications. Therefore, we developed an approach to extract function dependencies from a given component architecture (i.e., an implementation).

Through the analysis of the component architecture, we found function dependencies for more than 69% of all functions. Single functions had dependencies to more than half of all functions of their system. As a follow up, we investigated the found function dependencies in detail and discussed them with the developers. We assessed that almost 50% of our findings were considered as *plausible*, although the developers were *unaware* of them prior to the study. The study reveals that functions in productive multifunctional systems are highly interdependent and that current specification techniques address this issue only insufficiently. The study was carried out with MAN Truck & Bus AG and the BMW Group.

Requirements Engineering for Multifunctional Systems We present a comprehensive requirements engineering (RE) methodology that is based on a formal modeling theory for multifunctional systems.

We base our methodology on an artifact model that relates the semantic concepts of the modeling theory to model types that we use to describe them. The semantic relations between the concepts are reflected and expressed by the model types. The artifact model supports the stepwise modeling and formalization of functional requirements up to a system specification, which finally can be linked to the architecture of a system. We illustrate the role of the artifacts in a development process and the activities they enable and support. The methodology ensures the consistency of artifacts and allows for a comprehensive requirements tracing from informal requirements to architectures on different levels of abstraction especially with respect to the specification of function dependencies. Our main contribution is the definition of this methodology that allows a stepwise development of a formal and model-based specification for multifunctional systems. The methodology is evaluated in two case studies in which it is instantiated (1) in a scenario-driven RE context and (2) in a property-driven RE context. The first case study was conducted as part of a practical master's course at TU München and the second was carried out with Siemens in an industrial setting.

Finally, we highlight two central artifacts of the methodology, namely the *function documentation* and the *mode model*, and provide detailed instructions on how to systematically derive and structure these artifacts.

The proposed structure for function documentations is empirically grounded by an exploratory qualitative study in the context of automotive systems conducted at Robert Bosch GmbH. The conclusions of the study particularly address how function documentations should be structured and which information they should provide. We suggest documenting a function based on three levels of abstraction that are structured by a set of modes. The resulting function documentation template provides methodical guidance for the creation of a function documentation based on our proposed RE methodology.

The central role of modes in function specifications and in our methodology in general poses the question, how modes can be elicited systematically, and how large these mode models can get. In this thesis, we introduce three elicitation approaches for mode models, which we examine in an industrial case study conducted at MAN Truck & Bus AG, showing their feasibility and discussing differences between the elicited modes.

In the literature, approaches for a wide range of aspects and facets of multifunctional system specifications are proposed, including the specification of functional, safety, security, reliability, or timing requirements. This set of all types of requirements to be considered in a specification is large—beyond what can be covered in depth in a dissertation. In this work, we thus focus on functional requirements of a system as described by Sommerville [2011]: *"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."* Although the relevance of extra-functional requirements is often stressed, functional requirements are a premise for a valid system specification. Even more, most of the extra-functional requirements get a functional character once they are sufficiently broken down and refined (e.g., timing requirements add constraints to the functional requirements).

1.4 Approach: Model-based RE for Multifunctional Systems based on Functions and Modes

In this thesis, we follow a specific view onto specifications for multifunctional systems that is supported by a formal modeling theory proposed by Broy [2010b].

The application of the RE methodology introduced in this thesis results in a specification of a multifunctional system by following an iterative process along the two dimensions *modeling* and *formalization*. While modeling adds structure and provides a specific way of thinking about a specification, formalization adds precision and increases the potential for automation of development steps.

From a modeling point of view, requirements, in our approach, reflect a desired property of the system as observable at the interface between the system and its environment. Requirements are related to functions, which capture a set of requirements with a common purpose from a user's[2] point of view. The functions of a system are composed to composite functions and finally to a system specification. We call this the *function architecture* of a system. Functions may behave differently depending on the current state of a system. We model the state of a system in a *mode model*, which contains a (structured) set of modes. These modes can be referenced in the specification of functions as inputs or outputs. By this, function dependencies are modeled as interactions between functions via channels that transmit system modes.

From a formalization point of view, we embed the models used in the approach into a formal system modeling theory that represents a system as a stream processing function, which maps streams of input values to streams of output values. A specification is represented by a predicate over the set of stream processing functions.

1.5 Outline

Figure 1.1 gives an overview over the major contributions and structure of this thesis. Each box in the figure represents a contribution except for the formal system model described in Chapter 2, which is not a contribution of this thesis but serves as foundation. The contributions are structured with respect to their role in this thesis and

[2] A user can also be an external system (cf. the notion of an actor in UML [Fowler and Scott, 2000]).

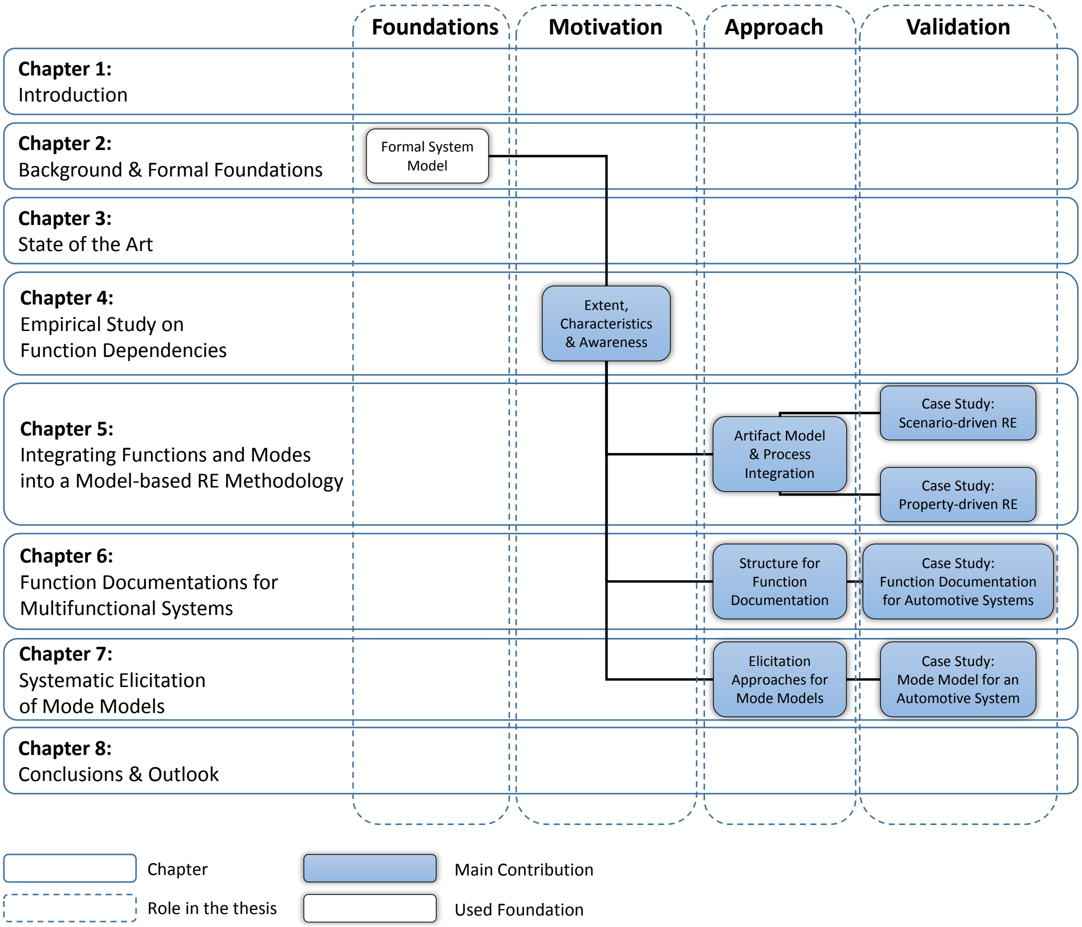

Foundations
Motivation
Approach
Validation
Chapter 1:
Introduction
Chapter 2:
Background & Formal Foundations
Formal System Model
Chapter 3:
State of the Art
Chapter 4:
Empirical Study on Function Dependencies
Extent, Characteristics & Awareness
Chapter 5:
Integrating Functions and Modes into a Model-based RE Methodology
Artifact Model & Process Integration
Case Study: Scenario-driven RE
Case Study: Property-driven RE
Chapter 6:
Function Documentations for Multifunctional Systems
Structure for Function Documentation
Case Study: Function Documentation for Automotive Systems
Chapter 7:
Systematic Elicitation of Mode Models
Elicitation Approaches for Mode Models
Case Study: Mode Model for an Automotive System
Chapter 8:
Conclusions & Outlook
Chapter
Role in the thesis
Main Contribution
Used Foundation

Figure 1.1: Overview over the main contributions and structure of this thesis.

related to chapters in which they are described. The solid lines between the contributions indicate how the contributions are related to each other. For example, the investigation of extent, characteristics, and awareness of function dependencies in Chapter 4 uses the formal system model as *foundation* to show the *motivation* for the *approaches* of Chapters 5 to 7, which are *validated* in the corresponding case studies.

Chapter 2 (Background and Formal Foundations) discusses the used terminology of this thesis and defines it with respect to a formal system model. Based on this system model, the scope of this thesis is defined in detail.

Chapter 3 (State of the Art) describes the current state of the art structured along the contributions of this thesis considering empirical work on function dependencies, specification techniques for multifunctional systems, approaches for function documentation, and elicitation approaches for mode models.

Chapter 4 (Empirical Study on Function Dependencies in Multifunctional Systems) presents an empirical study on the extent, characteristics, and awareness of function dependencies in existing multifunctional systems. It illustrates the challenges and motivates the approach of this thesis.

Chapter 5 (Integrating Functions and Modes into a Model-based RE Methodology) introduces our model-based requirements engineering methodology for the specification of multifunctional systems. It is based on an artifact model that defines the artifacts and modeling concepts used in the methodology. Furthermore, the methodology is instantiated and discussed in two case studies.

Chapter 6 (Function Documentations for Multifunctional Systems) introduces a documentation structure for functions that is based on modes and abstraction levels. The specific structure is grounded by a qualitative empirical study.

Chapter 7 (Systematic Elicitation of Mode Models for Multifunctional Systems) describes three approaches to elicit a mode model for a multifunctional system and discusses the approaches in the context of an industrial case study.

Chapter 8 (Conclusions and Outlook) summarizes this thesis by presenting its contributions, limitations and directions for future work.

Previously Published Material

Parts of the contributions presented in this thesis are based on previous publications:

[Vogelsang et al., 2012] Vogelsang, A., Teuchert, S., and Girard, J.: Extent and characteristics of dependencies between vehicle functions in automotive software systems. In: *Proceedings of the 4th International Workshop on Modeling in Software Engineering (MISE'12@ICSE)*, 2012.

[Vogelsang and Fuhrmann, 2013] Vogelsang, A. and Fuhrmann, S.: Why feature dependencies challenge the requirements engineering of automotive systems: An empirical study. In: *Proceedings of the 21st IEEE International Requirements Engineering Conference (RE'13)*, 2013.

[Vogelsang et al., 2014] Vogelsang, A., Eder, S., Hackenberg, G., Junker, M., and Teufl, S.: Supporting concurrent development of requirements and architecture: A model-based approach. In: *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD'14)*, 2014.

[Vogelsang, 2014] Vogelsang, A.: An exploratory study on improving automotive function specifications. In: *Proceedings of the 2nd International Workshop on Conducting Empirical Studies in Industry (CESI'14@ICSE)*, 2014.

[Böhm et al., 2014] Böhm, W., Junker, M., Vogelsang, A., Teufl, S., Pinger, R., and Rahn, K.: A formal systems engineering approach in practice: An experience report. In: *Proceedings of the 1st International Workshop on Software Engineering Research and Industrial Practices (SER&IPs'14@ICSE)*, 2014.

[Vogelsang et al., 2015] Vogelsang, A., Femmer, H., and Winkler, C.: Systematic elicitation of mode models for multifunctional systems. In: *Proceedings of the 23rd IEEE International Requirements Engineering Conference (RE'15)*, 2015

Chapter 2

Background and Formal Foundations

In this chapter, we introduce and explain the basic notions that are necessary to comprehend the content of this thesis. We especially discuss the notion of a function, which is essential for this thesis, and embed this notion into the context of multifunctional systems. For a precise definition of the terminology, we introduce a formal modeling theory and an architectural framework that is defined on top of that.

2.1 Terminology: What is a Function?

A central notion we use in this thesis is the notion of a *function*. The term function is associated with a great variety of meanings and interpretations in academia and industry. Additional terms that are often mentioned in this context are the terms *feature* and *service*. In the following, we informally describe a selection of different interpretations associated with these terms. In the remainder of this chapter, we will introduce a formal system modeling theory that we use to give a precise definition of the different interpretations and the interpretation we are going to use in this thesis.

As a major distinction between different definitions, we consider their focus on expressing *requirements*, *functional specifications*, or *design/implementation*. We list some of the common definitions categorized by their focus (cf. [Classen et al., 2008]).

Focus on Requirements: Kang et al. [1990] define a feature as *"a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems"*. This interpretation is prevalent in product line engineering approaches, where requirements (of a product line) are expressed by feature diagrams [Chen et al., 2005]. This definition also considers properties like color, a specific algorithm used, or special technical devices as features of a system. In these approaches, functions or features are generally not formalized any further and mainly serve the purpose of characterizing a certain product within a family of products.

Focus on Functional Specifications: Some definitions focus on functional characteristics of a system. Schätz [2008] defines, *"functions are capsules of behavior,* *defined by their (external) interface in terms of data and control flow [...]"* and Kang et al. [1998] refine their former definition of a feature to *"a distinctively identifiable functional abstraction that must be implemented, tested, delivered, and maintained"*. Similarly, Shaker et al. [2012] define a feature as *"a coherent and identifiable bundle of system functionality"*. These definitions aim at the use of the terms function or feature to be used as elements of a functional specification for a system. Broy [2010b] uses the term service for *"structuring of the functionality of multifunctional systems, with the emphasis on the specification phase of requirements engineering"*. All of these definitions are associated with a (partial) black-box view onto the system. These approaches agree on structuring a specification of a system into sub parts that contain extracts of the functionality, as it is perceived by the user or any other environmental system.

Focus on Design/Implementation: Many definitions consider functions, features, or services as elements of the design or implementation of a system. Liu et al. [2006] define a feature as *"an increment in program functionality"* and Apel et al. [2010a] define a feature as *"a structure that extends and modifies the structure of a given program in order to satisfy a stakeholder's requirement, to implement and encapsulate a design decision, and to offer a configuration option"*. Both definitions refer to elements of the implementing program. Some definitions specifically focus on the design/architecture of the implementing system. Jackson and Zave [1998] define a feature as *"an incremental unit of functionality in a DFC [Distributed Feature Composition] network"*. This definition is strongly influenced by the idea of decomposing a function into sub-functions, which may be arranged in a data flow or control flow network. This idea originates from classical functional decomposition approaches and structured analysis [DeMarco, 1979]. Decomposition is mainly influenced by steps of computation taken to fulfill desired functionality. Thus, this notion describes a white-box view onto a system by breaking it down into a network of functions (also known as function net or functional flow diagram).

As these examples show, it is important to have a precise definition of what is considered as a function. In the remainder of this thesis, we will associate the term function with an element that describes a black-box view onto a system to serve as (part of a) functional system specification. That means, a function describes the intent to use a system for a specific purpose, specified by a behavior that is observable at the boundary between the system and its operational context. Table 2.1 gives an overview over the most important terms used in this thesis and provides an informal description of them. Additionally, we associate related terms used in other publications with them.

By relating the central terms to each other, we draw a big picture of our *semantic model* of requirements engineering concepts for multifunctional systems, illustrated in Figure 2.1. The elements of the semantic model do not refer to any specific description technique. Instead, we use this semantic model to understand and relate the *content* that is expressed by different description techniques.[1]

[1] The differentiation between the semantics (content), the syntax (description technique), and the representation of an artifact may sound hairsplitting or trivial. However, in the context of *artifact orientation*, it is a relevant research topic [Böhm and Vogelsang, 2013; Méndez Fernández et al., 2010].

Table 2.1: Central terms of this thesis.

Term	Description	Related terms
Multifunctional system	A system that offers a variety of different *functions* to its environment.	Multifeatured system [Batory et al., 2004]
(Functional) Requirement	A (behavioral) property a *multifunctional system* must fulfill.	Feature [Kang et al., 1990]
Function	An intent of using a *multifunctional system* in a specific usage context for a certain purpose.	Use case [Cockburn, 2001], Feature [Kang et al., 1998; Zave, 1999]
Function specification	A (set of) interaction patterns specifying a *function*'s behavior.	Use case [Cockburn, 2001], Feature [Shaker et al., 2012], Usage [Jackson and Zave, 1998], Service [Broy, 2010b]
Function architecture	A (structured) set of (dependent) *functions* specifying the behavior of a *multifunctional system*.	Feature model [Kang et al., 1998], Service hierarchy [Broy, 2010b], Service [Bouma and Velthuijsen, 1994], Use case diagram [OMG, 2011]
Function dependency	Behavioral interaction between *functions* specifying desired behavior of a *multifunctional system*.	Feature interaction [Zave, 1999], Lifter [Prehofer, 1997], Derivative [Liu et al., 2006]
Mode	An operational state of a *multifunctional system* or its environment. *Function dependencies* are modeled via modes.	Mode [Dietrich and Atlee, 2013]
Mode model	A (structured) set of *modes* specifying the possible dependency interactions in a *function architecture*.	Statechart [OMG, 2011]
Component	Part of a solution structure that realizes/ implements *functions* of a *multifunctional system*.	Feature box [Jackson and Zave, 1998], Feature module [Batory et al., 2004], Feature [Apel et al., 2010a]
Component architecture	A (structured) set of (communicating) *components* describing the solution structure of a *multifunctional system*.	Implementation model [Kästner et al., 2009], DFC network [Jackson and Zave, 1998]

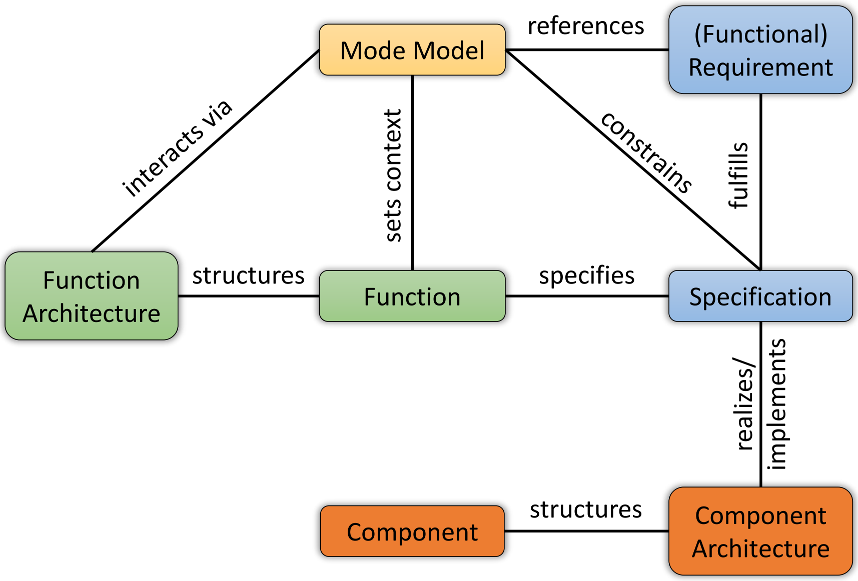


Figure 2.1: Semantic model of requirements engineering concepts for multifunctional systems.

One aspect of the model that may need some additional explanation is the differentiation between a *function* and its *specification*. The rationale behind this separation in our model is that a function can be seen as an intent to use a system for a specific purpose, independent of *how* to use the system. Consider the function "Cruise Control" of a car. This function describes the intent to use the system "car" for holding the vehicle's speed on a desired level. A *function specification* specifies *how* that system is used in terms of interactions with its environment. For example, a *function specification* may state that the driver first has to activate the "Cruise Control", which is confirmed by a system response, and after that, the driver can increase or decrease the desired speed based on the current speed at function activation. In general, many function specifications, i.e., scenarios of use, are imaginable for one function. There are terms and concepts in literature that focus on the intent and purpose (e.g., feature [Kang et al., 1998; Zave, 1999]) and there are terms that focus on function specifications (e.g., usage [Jackson and Zave, 1998], feature [Shaker et al., 2012]). Use cases, as introduced by Cockburn [2001], are a mixture of both because, on the one hand, they emphasize the purpose of a use case while, on the other hand, they are supplemented by concrete usage scenarios. In our thesis, we aim at a functional specification of multifunctional system. Therefore, we sometimes use the concepts *function* and *function specification* interchangeably. For example, if we speak of the interface of a function, we actually mean the interface that belongs to a *function specification* that specifies a *function*. However, we think that it is valuable to recognize the difference between these notions.

In the following section, we provide formal definitions for all of the terms and their relations by embedding them into a formal system modeling theory. In Chapter 5 of this thesis, we introduce model types that we use to develop, document, and analyze the concepts of this semantic model.

2.2 Formal Foundation

For the purpose of precise definitions, we ground the semantic concepts of this thesis on the formal system modeling theory FOCUS [Broy and Stølen, 2001], which provides formal concepts to describe a system, and an architectural framework [Broy et al., 2012], which defines viewpoints that describe a system in different phases of a development process. In the following, we briefly introduce the system modeling theory and the architectural framework. For more information on the modeling theory, please refer to Broy's earlier work [Broy, 2007, 2010a,b; Broy and Stølen, 2001].

2.2.1 System Modeling Theory

In the formal system modeling theory FOCUS, a system is described by its interface. An interface description contains syntactic and behavioral information that characterize a system. The theory provides formal notions of interface description, interface composition, and interface specification.

Messages and Streams

To describe the interface of a system, we first need to introduce messages and streams. A stream is an infinite sequence of elements of a given set. In interactive systems, streams are built over sets of messages or actions. Streams are used that way to represent communication histories. Let X be a given set of messages. X^∞ denotes the set of infinite sequences over X (represented by the total mappings $\mathbb{N} \to X$). A stream s is an element of this set: $s \in X^\infty$.

Interface Description

Types are useful concepts to describe system interfaces. We work with a simple notion of types where each type is a set of data elements. These data elements are used as messages. Let a set T of types be given. Then, the universal set of messages X is given by the union of all types:

$$X = \bigcup_{t \in T} t$$

In FOCUS, a *typed channel* is an identifier for a sequential directed communication link for messages of that type. By C, we denote a set of typed channel names. We assume that a type assignment for the channels in the set C is given as follows:

$$type : C \to T$$

Given a set C of typed channels, a *channel valuation* is an element of the set $\vec{C}$ that is defined as follows:

$$\vec{C} = \{x : C \to X^\infty : \forall c \in C : x(c) \in (type(c))^\infty\}$$

A channel valuation $x \in \vec{C}$ associates a stream of elements of type $type(c)$ with each channel $c \in C$. This way the channel valuation x defines a *channel history* for the channels in the set C. If we are only interested in the channel valuation of a subset $C' \subseteq C$ of channels, we use the *restriction* operator $x|C'$ with:

$$x|C' = y \quad \text{such that} \quad \forall c \in C' : y(c) = x(c)$$

In our system modeling theory a (discrete) system consists of

A syntactic interface: The syntactic interface defines the input and output information that the system is able to sense and produce, and structures them into a set of (typed) input channels I and a set of (typed) output channels O. We denote the syntactic interface of a system by the term $(I \blacktriangleright O)$.

An interface behavior: The behavior of a system is described by means of relations between streams associated to the syntactic interface. The *interface behavior* F of a system is then represented by a mapping

$$F : \vec{I} \to \wp(\vec{O})$$

where $\wp(\vec{O})$ denotes the power set of all output channel histories. This describes the (nondeterministic) behavior of a system by relating streams of input messages to streams of output messages.

Interface Composition

A fundamental operation in the system model is the composition of interfaces (e.g., to describe the composition of systems). Given two interfaces with disjoint sets of output channels ($O_1 \cap O_2 = \emptyset$) and interface behaviors

$$F_1 : \vec{I_1} \to \wp(\vec{O_1}), \qquad F_2 : \vec{I_2} \to \wp(\vec{O_2})$$

we define the parallel composition with feedback by the interface behavior

$$F_1 \otimes F_2 : \vec{I} \to \wp(\vec{O})$$

where the syntactic interface is specified by the equations

$$I = (I_1 \cup I_2) \setminus (O_1 \cup O_2), \qquad O = O_1 \cup O_2$$

In this definition of interface composition, the internal channels are also part of the composite interface, i.e., no *channel hiding*[2].

The composite interface behavior is specified by the following equation ($y \in \vec{C}$, where the set of channels C is given by $C = I_1 \cup I_2 \cup O_1 \cup O_2$):

$$(F_1 \otimes F_2)(x) = \{y|O : y|I = x \wedge y|O_1 \in F_1(y|I_1) \wedge y|O_2 \in F_2(y|I_2)\}$$

Here, y denotes a valuation of all the channels in C of F_1 and F_2. The formula essentially says that all the streams on the output channels of the composite system $F_1 \otimes F_2$ are feasible output streams of the systems F_1 and F_2.

[2] A definition of interface composition with *channel hiding* can easily be derived by restricting the composite output stream to the external channels $O' = O \setminus (I_1 \cup I_2)$ [Broy, 2010a].

Interface Specification

There are several ways and techniques of specifying interfaces. We relate the different specification techniques to three basic views onto a system, which all contain information about the syntactic interface as well as the interface behavior of a system.

Interface view: Specification techniques of the interface view describe the system by a set of input/output channels and a logic formula defined over these channels. Specification techniques of this view resemble the basic system model most directly. Examples for specifications of this view are temporal logic specifications such as LTL or CTL.

Architecture view: Specification techniques of the architecture view describe the system by a network of communicating subsystems, which again are considered as systems themselves. The syntactic interface of a system as well as its interface behavior in this view is derived from the composition of the subsystems. Examples for specifications of this view are dataflow diagrams or process algebras.

State view: Specification techniques of the state view describe the system by a set of states and transitions between them. Transitions are triggered by events and may produce actions. The syntactic interface of the system in this view is determined by the events and actions of the transitions. The interface behavior is defined by the set of possible event/action traces that result from traversing the states by the transitions. An example for a specification of this view is a state machine.

2.2.2 Architectural Framework

The formal system modeling theory can be used in different phases of the development process to describe a system from different points of view, addressing different concerns. This follows the idea of separating concerns into different viewpoints [ISO/IEC/IEEE, 2011a]. In the publicly funded research project SPES, we developed an architectural framework that structures the system development into four basic viewpoints: requirements, functional, logical, and technical [Broy et al., 2012]. Each viewpoint contains a set of concepts, which are customized towards the concerns of the different viewpoints. All concepts rely on the system modeling theory and have the purpose to describe and specify system interfaces from a specific point of view. In the context of this architectural framework, we can give precise definitions of the different concepts we use in our semantic model (see Figure 2.1).

Requirements Viewpoint

The requirements viewpoint contains concepts that are used to model and formalize single requirements. A requirement is a property the system must fulfill. With respect to the system modeling theory, a requirement describes desired observations of behavior between a system and its environment. Therefore, the specification of a requirement is expressed as a predicate p defined over the channel histories of a system:

$$p : \vec{I} \times \vec{O} \rightarrow \mathbb{B}$$

While most requirements are initially expressed in natural language, the model types of the requirements viewpoint allow a stepwise modeling and formalization of certain types of requirements to a point, where the requirement is expressed as formal predicate. This provides a formal and model-based representation of requirements.

Functional Viewpoint

In the functional viewpoint, we perform the step from a formal and model-based representation of requirements to a system specification. The functional viewpoint is concerned with a purely functional view onto the system. In this viewpoint, (formalized) requirements are related to functions of a system, which group the requirements according to functional concerns.

Function and Function Specification A function describes the intent to use a system in a specific *usage context* for a certain *purpose*. We distinguish functions by giving them a unique name. The interactions that occur between a system and its environment during the usage of a function are specified by a function specification. A *function specification* describes a pattern or a set of patterns of using a system for a certain purpose. These interaction patterns refer to events of sending and receiving messages. Thus, we can formally specify functions by their interfaces (see Section 2.2.1): A function has a unique name f and is specified by a syntactic interface $(I_f \blacktriangleright O_f)$ and an interface behavior:

$$F_f = \vec{I_f} \rightarrow \wp(\vec{O_f})$$

This interface behavior is, in general, a partial mapping. Partiality here means that a function is specified only for a subset of its input histories, according to its syntactic interface. This subset is called the *function domain* [Broy, 2010b; Broy et al., 2007b]. We aim at a functional specification of multifunctional system (cf. Section 2.1). Therefore, we sometimes use the concepts *function* and *function specification* interchangeably. For example, if we speak of the interface of a function, we actually mean the interface that belongs to a *function specification* that specifies a *function*. A similar view is also taken in the approaches of Shaker et al. [2012], Schätz [2008], and Broy [2010b].

Function Architecture In the functional viewpoint, functions of a (multifunctional) system are arranged in a *function architecture*, in which atomic functions are composed to function groups and finally to the functional specification of the system under development.

A function architecture is represented by a finite set of functions Fun, a mapping $sub : \mathit{Fun} \rightarrow \wp(\mathit{Fun})$ that represents a sub-function relation, and a mapping $dep : \mathit{Fun} \rightarrow \wp(\mathit{Fun})$ that represents a dependency relation. For a function $f \in \mathit{Fun}$ the set $sub(f)$ is called its sub-function family. The functions f in a function architecture without sub-functions (i.e., $sub(f) = \emptyset$) are called atomic functions of the architecture. All other functions are composite functions. A dependency relation between functions is only allowed for functions that are not in an ancestral relation, i.e., $\forall f, g \in \mathit{Fun} : g \in \mathit{sub}^*(f) \Rightarrow g \notin \mathit{dep}(f)$, where sub^* is the transitive-reflexive closure of sub.

We apply the interface composition operator (see Section 2.2.1) to define the syntactic interface and the interface behavior of a composite function. Thus, the interface behavior of a composite function f with sub-function family $\mathit{sub}(f)$ is defined by

$$F_f = \bigotimes_{g \in \mathit{sub}(f)} F_g$$

The syntactic interface and the interface behavior of the entire function architecture are defined by the root function f_r, i.e., the function that is not contained in the sub-function family of any function in the function architecture ($\forall f \in \mathit{Fun} : f_r \notin \mathit{sub}(f)$).

The dependency relation dep represents functional dependencies. In the formal system modeling theory, functional dependencies are specified by communication channels between interfaces. Therefore, the following must hold for functions $f, g \in \mathit{Fun}$ with interfaces $(I_f \blacktriangleright O_f)$ and $(I_g \blacktriangleright O_g)$:

$$g \in \mathit{dep}(f) \Rightarrow O_f \cap I_g \neq \emptyset$$

The dependency relation is interpreted by communication channels between functions. We call these internal communication channels *mode channels*, while we call the other channels of a function interface *primary channels*.

Modes and Mode Model Internal channels between functions of a function architecture have a special role. They are called *mode channels* because the type that defines the messages transmitted over these channels is called a *mode*. A mode describes (a part of) the operational state of a multifunctional system or its environment. The motor of a car, for example, may be characterized by a mode $\mathit{Operation}$ that has mode values Off, $\mathit{Starting}$, and $\mathit{Running}$. A multifunctional system may be characterized by a whole set of modes. The usage context of a function is characterized by modes (e.g., "While *driving*, I want the vehicle to hold its speed on a desired level"). Modes are structured in a *mode model*. We describe a mode model formally by adopting the formal description of statecharts [Harel, 1987] as described by Eshuis [2009]:

A mode model is represented by a pair $\mathit{MM} = (M, T)$, where M is a set of modes and $T \subseteq M \times M$ is a set of mode transitions. Function $\mathit{children} : M \rightarrow \wp(M)$ defines for each mode m its immediate submodes. There are several kinds of modes. If m has no children, so $\mathit{children}(m) = \emptyset$, then m is a BASIC mode. Otherwise, m is composite. A composite mode is either an OR mode or an AND mode. Function $\mathit{kind} : M \rightarrow \{\mathit{AND}, \mathit{OR}, \mathit{BASIC}\}$ assigns to each mode its kind. Function $\mathit{default} : M \rightarrow M$ identifies for each OR mode m one of its children as the default mode: $\mathit{default}(m) \in \mathit{children}(m)$.

Semantically, a mode model specifies sequences of mode configurations. A mode configuration Cfg is a set of modes $\mathit{Cfg} \subseteq M$ that represents a valid global state of the mode model. If, for example, an OR mode is part of a mode configuration, then exactly one of its children is also part of the configuration. Whereas, for each AND mode in a mode configuration, all of its children are also part of the configuration. The mode transitions specify the valid transitions between mode configurations. A detailed definition of mode configurations and transitions between those is given by Eshuis [2009]. We describe the set of (infinite) sequences of mode configurations specified by a mode model by Cfg^{∞}.

The valid sequences of mode configurations defined by a mode model constrain the possible mode channel histories in a function architecture. To describe this formally, we need to define the set of mode types MT for a given mode model by $MT = \{children(m) : kind(m) = OR\}$. The set of mode types is equivalent to a set of (data) types as used for input/output channels of functions in a function architecture.

Let f be a function in a function architecture with a syntactic interface $(I \blacktriangleright O)$ that consists of primary and mode channels, i.e.,

$$I = I_p \cup I_m, \qquad O = O_p \cup O_m$$

and an associated interface behavior $F_f : \vec{I} \rightarrow \wp(\vec{O})$. Let MM be a mode model that specifies a set of mode types MT and sequences of valid mode configurations Cfg^{∞}. Function f is called consistent with mode model MM if the following holds:

1. All mode channels of f have a mode type: $\forall m \in I_m \cup O_m : type(m) \in MT$.
2. The interface behavior of f does not violate the valid mode configurations specified by MM:
$$\forall x \in \vec{I} : F_f(x)|O_m \in \mathit{Cfg}^{\infty}|O_m$$

Note that the composition of two function interfaces that are consistent with a mode model does not necessarily result in a composite interface that is also consistent with the mode model. That is, however, not surprising because the purpose of the mode model is exactly to constrain the possible behavior that results from integrating functions into one system. Therefore, a function architecture is consistent with a mode model if its root function is consistent with the mode model.[3]

Function, function architecture, and mode model as concepts of the functional viewpoint together serve as a functional specification for a system.

Logical Viewpoint

The concepts of the logical viewpoint describe a system with a focus on the logical realization structure by means of communicating logical components. In contrast to the function architecture, the *component architecture* that is defined in the logical viewpoint is not solely structured with respect to functionality but in terms of architectural design. Here, also aspects like the organizational structure, dependability, maintainability, and reusability play an important role.

With respect to the formal system modeling theory, logical components are described similarly to functions. A logical component is described by a syntactic interface $(I_c \blacktriangleright O_c)$ and an associated interface behavior $F_c = \vec{I_c} \rightarrow \wp(\vec{O_c})$. A network of logical components forms the component architecture of a system. The component architecture itself forms a logical component with a syntactic interface and an interface behavior that reflects the interface of the system to its environment. The interface

[3] This definition relies on the composition without channel hiding as introduced in Section 2.2.1.

and behavior of the component architecture results from the composition of its logical components as described in Section 2.2.1.

The difference between a function architecture and a component architecture is purely methodological, i.e., we use the same formal concepts to describe functions and components. The description of this methodological difference and how it is reflected in a development process is part of this thesis and is described in Chapter 5.

The composition of the logical components must refine the functionality specified by the composition of the functions from the function architecture. Let $F_f : \vec{I_f} \to \wp(\vec{O_f})$ be the interface behavior of a function architecture composed of functions $f_1, \ldots, f_n$ and $F_c : \vec{I_c} \to \wp(\vec{O_c})$ be the interface behavior of a component architecture composed of logical components $c_1, \ldots, c_m$, then the following proposition must hold:

$$\forall x \in \vec{I_f} : F_c(x) \subseteq F_f(x)$$

The formula states that the component architecture is a refinement of the behavior specified by the function architecture. In the remainder of this thesis, we will see that, in some cases, it is necessary to "translate" the stream of a function to a stream of a component due to different levels of abstraction for example (cf. *interaction refinement* [Broy, 2010a]).

The notions of the terms function or feature that focus on design/implementation (cf. Section 2.1) can be defined and formalized in the context of the component architecture of a system. For example, what Jackson and Zave [1998] described as *feature box* can be seen as an atomic logical component, i.e., a logical component that is not further decomposed. A network of feature boxes (atomic logical components) may describe the functional decomposition of a *black-box function* into processing steps. In the course of architectural design, atomic logical components may be grouped or even merged into more comprehensive logical components that finally build the architecture of a system. Such a comprehensive logical component can be seen as *"an incremental unit of functionality"*, which is the definition of *feature* as given by Zave [2003].

Technical Viewpoint

The purpose of the technical viewpoint is to provide concepts of the system with a focus on the target execution platform. In a deployment mapping, the logical components of the component architecture are assigned to technical execution units, which execute the specified behavior of the logical component.

The concepts of this viewpoint are not in the scope of this thesis, although also in the context of technical execution platforms the term function is commonly used. For example, when a technical component is used for one specific purpose, some people give the function the name of the technical component.

2.3 Scope of this Thesis

The scope of this thesis is a methodology for the creation and analysis of models residing in the requirements and functional viewpoint as well as its relation to the models of the component architecture (logical viewpoint). The thesis provides model types to be used in the requirements and functional viewpoint along with analysis techniques. The resulting models serve as functional specification of a multifunctional system. How the functionality is realized in an implementation is not part of this thesis. We focus on functional requirements and aim at the specification of multifunctional systems.

2.4 Modeling vs. Formalization

The approach taken in this thesis utilizes modeling and formalization as means to reduce complexity and increase precision in the specification of multifunctional systems. However, it is important to notice that these two dimensions are independent from each other in general. While modeling structures the development artifacts and applies a common terminology, formalization adds precision and reduces the room for interpretation.

Although the methodology described in this thesis exploits both dimensions, it is possible to apply the methodology by focusing only on one dimension. For example, in an industrial context it might be interesting to follow only the modeling dimension of the methodology by structuring artifacts according to the model types of the methodology but still describing the specific elements by natural language text.

Chapter 3

State of the Art

This chapter summarizes existing work in the research area of specifications for (multifunctional) systems. More specifically, it summarizes work on extent and characteristics of function dependencies and on approaches for the specification of multifunctional systems. Each section summarizes existing work, outlines open issues, and points to the chapters in this thesis that contribute to their resolution. The structure of this chapter reflects the organization of this thesis.

Section 3.1 outlines work on the extent, characteristics, and impact of function dependencies. The presented work provides evidence that function dependencies (also called feature interactions) are numerous in existing systems and that they are a major source of failures. The contributions of our thesis presented in Chapter 4 support this evidence and additionally indicates that developers are unaware of function dependencies in most cases.

Section 3.2 outlines work on specification techniques for multifunctional systems and, if existent, their integration in requirements engineering approaches. We show that existing specification approaches lack an explicit specification of function dependencies (especially with respect to behavior), do not rely on a seamless modeling theory, or focus on implementation rather than on specification. In Chapter 5, we present an approach that employs (behavioral) function dependencies as explicit elements of a specification that is based on a formal modeling theory.

Section 3.3 outlines work on creating a comprehensive function documentation including proposed artifact structures and templates. We show that existing work recognizes the need for structure and abstraction in function documentations, which is also reflected by our study presented in Chapter 6. However, currently proposed templates and approaches do not consider both aspects at the same time.

Section 3.4 outlines work on the specification of systems based on states/modes. We show that these approaches do not provide any guidance on how to elicit states/ modes. In Chapter 7, we present three systematic approaches to elicit a mode model as a prerequisite for mode-based function specifications.

3.1 Extent, Characteristics, and Impact of Function Dependencies

In the following, we report on work related to the extent and characteristics of function dependencies and their impact on the development process. The contribution of Chapter 4 builds up on and extends this state of the art.

3.1.1 Extent and Characteristics of Function Dependencies

Function dependencies have been extensively investigated in the telecommunication domain [Calder and Magill, 2000]. In this context, a function dependency is called *feature interaction* and is defined by Zave [1999] as *"some way in which a feature or features modify or influence another feature in defining overall system behavior."* A feature, here, is *"an increment of functionality, usually with a coherent purpose."* [Zave, 1999]

Apel et al. [2013b] explored feature interactions in real-world systems and characterized them by two dimensions: order and visibility. The order of a feature interaction is defined as the minimum number of features (minus one) that need to be activated to trigger the interaction. For the visibility, the authors distinguish between *external* and *internal* feature interactions. External feature interactions may appear at the level of the externally-visible behavior. They are subdivided into *functional* interactions, which address interactions violating the functional specification of a system and *non-functional* interactions, which address interactions influencing non-functional properties (e.g., performance, memory consumptions, or energy consumption). Internal feature interactions, on the other hand, may appear at the level of the internal properties of a system. They are subdivided into *structural* interactions, which can be detected by static analysis of the syntactic program structure and *operational* interactions, which can only be detected by more sophisticated analyses (e.g., control or data flow analysis). In their article, Apel et al. [2013b] give preliminary results considering the detection and classification of feature interactions in four real-world systems: LINUX, BUSYBOX, GCC, and APACHE. Feature interactions occurred in all systems, and the authors found interactions of all kinds, including structural, operational, functional, and non-functional interactions. The internal feature interactions outnumbered the external feature interactions found.

Kästner et al. [2009] examined the extent of the *optional feature problem* in two case studies from the domain of embedded database systems. The optional feature problem describes a common mismatch between variability intended in the domain and dependencies in the implementation. The optional feature problem occurs if two (or more) optional features are independent in a domain, but are not independent in their implementation. For a closer analysis, the authors first distinguish between a feature model and an implementation model in software product line development. A feature model (also known as domain model or product line variability model) describes the features of a domain or software product line and their relationships. An implementation model (also known as software variability model or family model) describes implementation modules (such as components, plug-ins, aspects, or feature modules) and their relationships. Feature model and implementation model are linked, so that for a given feature selection the according implementation modules can be composed. In both case studies, the authors found significantly more dependencies in the implementation models than in the feature models. For the first case study, they extracted 38 features with 16 domain dependencies but with 53 implementation dependencies. In the second case study, they extracted 24 features with only 8 domain dependencies but with 78 implementation dependencies. These results show that there is a considerable difference in handling function dependencies on an implementation/architectural level and on a level of functions.

Relation to our thesis In this thesis, we focus on *external functional* feature interactions [Apel et al., 2013b] and call them function dependencies. The approach we present in Chapter 5 aims at explicitly modeling these function dependencies as part of a system specification. In Chapter 4, we will report on a study, in which we investigate *structural* (i.e., internal) interactions in an automotive system to derive function dependencies. The results support the presumption of Apel et al. [2013b], who assume a relation between internal and external feature interactions: *"We [the authors] believe that there may be systematic correlations between externally-visible and internally-visible interactions, which is a major motivation for our endeavor to explore and understand the nature of feature interactions."*

One conclusion of the study on the optional feature problem [Kästner et al., 2009] is that dependencies on an implementation/architectural level should be separated and handled differently from dependencies on a level of functions. We support this conclusion from a different angle by the results of the study presented in Chapter 4. In this study, we show that specifying dependencies solely on an implementation/ architectural level leads to a high chance of missing dependencies on the level of functions.

3.1.2 Impact of Function Dependencies on Integration Failures

Cataldo and Herbsleb [2011] present a study on factors leading to integration failures in global feature-oriented development. They analyzed a large-scale project that implemented 1195 features in a software system. Their analyses revealed that cross-feature interactions, measured as the number of architectural dependencies between two product features, are a major driver of integration failures (detected by failing integration tests).

Figure 3.1 illustrates the relation between the number of cross-feature dependencies and the probability of having integration failures. The figure additionally opposes this relation for geographically distributed and collocated teams (*GSD* is a dichotomous variable where 1 indicates that the feature team members were located in different development sites; otherwise, it is set to 0). As the number of cross-feature dependencies increases (x-axis), we observe that there is a point (values > 4 in the log-transformed measure on the x-axis) at which the probability of integration failures (y-axis) increases significantly faster when the teams that worked on a pair of features are collocated than when feature teams are geographically distributed. It is important to point out that for levels of the cross-feature dependencies measure below 5.5, the probability of having integration failures is more than double for the GSD case than for the collocated case. However, when features are highly interre-

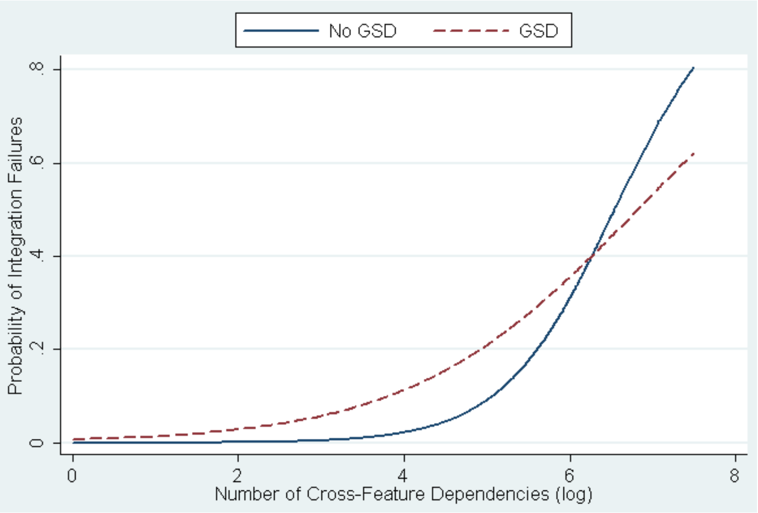


Figure 3.1: The interplay between the geographic distribution of the feature teams and cross-feature dependencies [Cataldo and Herbsleb, 2011].

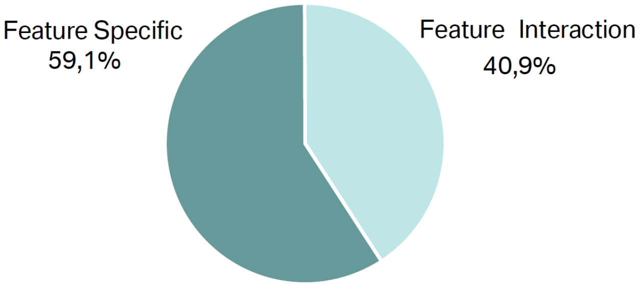


Figure 3.2: Faults resulting from feature interaction [Benz, 2010].

lated (value > 6), the results show that the impact of cross feature dependencies is lower when feature teams are geographically distributed. One possible explanation the authors give for this result is that the work practices developed by collocated teams might allow them to handle certain levels of interdependence between features very well. For example, dependencies might be handled more informally because the engineers are physically collocated. However, beyond a certain point, those work practices fail to adequately identify and manage the dependencies between features. On the other hand, distributed teams are always at a disadvantage and, by recognizing such condition, they might develop different work practices to manage dependencies that help them cope better with high levels of interdependence.

Benz [2010] reports on feature interactions in infotainment systems of automobiles. During the development of an infotainment system, he reports that a large fraction of the faults resulted from feature interactions. Figure 3.2 shows the distribution of severe faults that occurred during development with respect to whether they were feature specific or the result of feature interactions. The figure shows that more than 40% of all severe faults occurred in feature interaction scenarios. The study has been performed as part of a diploma thesis at BMW. During the study, the examined system was divided into its different features. All faults that result from the interaction between different features have been classified as feature interaction faults.

Relation to our thesis The extraction of cross-feature dependencies as presented by Cataldo and Herbsleb [2011] is comparable to our extraction of function dependencies used in Chapter 4. In their study, they could show with statistical significance that the higher the number of function dependencies a pair of functions have, the higher is the likelihood of integration failures to occur. In our study, we have not investigated the relation between function dependencies and integration failures but we observed a high number of function dependencies in productive multifunctional systems and showed that developers are, in most cases, not aware of them. Together with the results from the study of Cataldo and Herbsleb [2011], this leads to the conclusion that many severe faults are due to function dependencies. This corresponds to the observations presented by Benz [2010].

3.2 Specification Techniques for Multifunctional Systems

In the following, we outline work on specification techniques for multifunctional systems and, if existent, their integration into requirements engineering approaches. The contributions of Chapter 5 build up on and extend this state of the art.

3.2.1 Requirements Modeling

Many requirements modeling approaches are based on the RE reference model presented by Jackson and Zave [1995] and Gunter et al. [2000]. Their model is a framework for talking about key artifacts, their attributes, and relationships at a general level. The model is based on five artifacts broadly classified into groups that pertain mostly to the system versus those that pertain mostly to the environment. The artifacts are characterized by the *phenomena* they are talking about. The phenomena of a specific artifact are called *designations*. Designations identify classes of phenomena, typically states, events, and individuals, in the system and the environment and assign formal terms (names) to them. Gunter et al. [2000] illustrate the framework by a small example of a warning system that notifies a nurse if the patient's heartbeat stops. The system has a sensor to detect sound in the patient's chest and an actuator that can be programmed to sound a buzzer based on data received from its sensor. Additionally, we assume that there is always a nurse close enough to the nurse's station to hear the buzzer, and that if the patient's heart has stopped, the sound from the patient's chest falls below a threshold for a certain time.

Designations are structured into the following categories:

e_h: Phenomena that are hidden from the system and controlled by the environment (e.g., the nurse and the heartbeat of a patient)

e_v: Phenomena that are visible to the system and controlled by the environment (e.g., sounds from the patient's chest)

s_v: Phenomena that are visible to the environment and controlled by the system (e.g., the buzzer at the nurse's station)

s_h: Phenomena that are hidden from the environment and controlled by the system (e.g., internal representation of data from the sensor)

These classes of phenomena characterize the five artifacts employed by the reference model. The artifacts are:

World (W): Describes domain knowledge that provides presumed environment facts. The world model describes domain assumptions that are indicative, i.e., they indicate facts. The world model is exclusively described by designations that are visible to the environment (e_h, e_v, s_v). For example, a world model may state that there is always a nurse close enough to the nurse's station to hear a buzzer.

Requirements (R): Indicate what the customer needs from the system, described in terms of its effect on the environment. They are optative descriptions, i.e., they express how the world should be once the system is deployed. Requirements are exclusively described by designations that are visible to the environment (e_h, e_v, s_v). For example, a requirement may demand a warning system to notify a nurse if the patient's heartbeat stops.

Specification (S): Provides enough information for a programmer to build a system that satisfies the requirements. The specification is exclusively described by designations that are visible to the system and the environment (e_v, s_v). For example, a specification may state that if the sound, the sensor detects, falls below the appropriate threshold, then the system should sound the buzzer.

Program (P): Implements the specification using the programming platform. A program is exclusively described by designations that are visible to the system (e_v, s_v, s_h). For example, a program may implement to sound a buzzer based on data received from its sensor.

Machine (M): Provides the basis for programming a system that satisfies the requirements and specifications. A machine is exclusively described by designations that are visible to the system (e_v, s_v, s_h). For example, a machine may have a sensor to detect sound in the patient's chest and an actuator capable of sounding a buzzer.

Figure 3.3 illustrates the five artifacts of the model and their corresponding designations. A central proof obligation of the reference model is the following relationship:

$$W, S \vdash R$$

This means that the system satisfies the requirements if S combined with W entails R. The specification S in this case is to stand proxy for the program with respect to the requirements. If S properly takes W into account in saying what is needed to obtain R, and P is an implementation of S for M, then P implements R as desired.

Relation to our thesis The approach presented in this thesis and its formal foundations can be related to the RE reference model: Functions, function architecture and mode model as introduced in Chapter 2 are parts of the *specification* artifact as their purpose is to specify the relation between environmental stimuli and system reactions. A function is specified by relating phenomena e_v and s_v. Both types of phenomena may refer to a mode from the mode model. That means the mode model captures phenomena that are visible to the system and the environment and may be controlled by both.

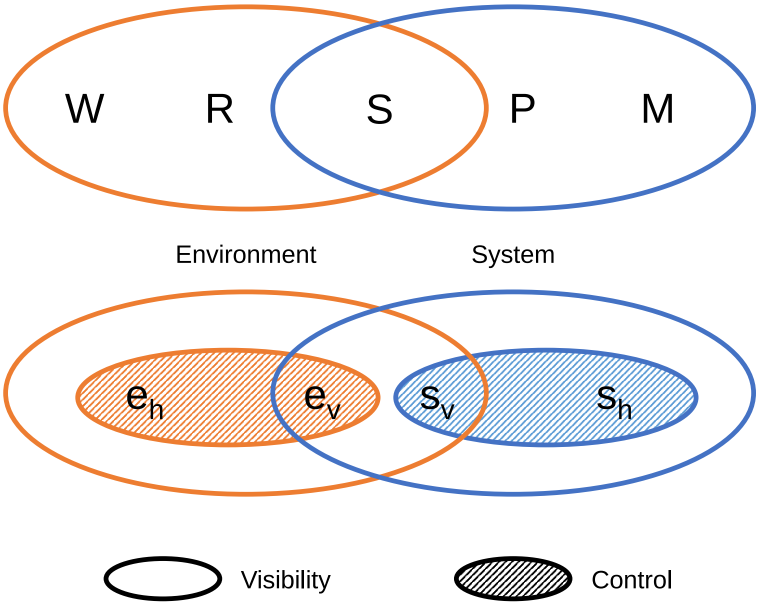


Figure 3.3: Five software artifacts with visibility and control for designated terms [Gunter et al., 2000].

Besides the approach presented in this thesis, there are other requirements modeling approaches that follow the Jackson and Zave RE reference model. For example, KAOS [van Lamsweerde, 2009] is a requirements engineering methodology that covers identification of the business requirements, of the requirements, of the responsible agents and, if needed, of the behaviors they need to conform to satisfy the requirements. KAOS proposes four related artifacts to model the requirements for a system (see Figure 3.4). The *goal model* is a directed graph of goals that are decomposed into subgoals that enumerate all the cases that must be covered to fulfill the parent goal. Goals that are not further refined into subgoals (i.e., leafs of the goal model) must be placed under the responsibility of an agent, which is part of the environment or the system. The leaf goals represent either an expectation (when placed under responsibility of an environment agent), a requirement (when placed under responsibility of a system agent), or a domain property (descriptive assertion about objects in the environment). Goal modeling is a common technique also used in other contexts to justify requirements by linking them to higher-level goals. Besides the goal modeling technique used in KAOS, there are also other goal modeling notations (e.g., iStar [Yu et al., 2011] or GSN [Kelly and Weaver, 2004]). In KAOS, the agents that are linked to goals are modeled in the *responsibility model*. The *operation model* describes all the behaviors that agents need, in order to fulfill their requirements. Behaviors are expressed in terms of operations performed by agents. Operations work on objects (defined in the *object model*): they can create objects, trigger object state transitions, and activate other operations (by sending an event). Concerned objects are connected to the operations by means of input and output links. Events can be external or produced by operations. They may start (cause) or stop operations. The notion of an operation in KAOS comes closest to the notion of a function in our approach. However, KAOS does not support specification or analysis of function (operation) dependen-

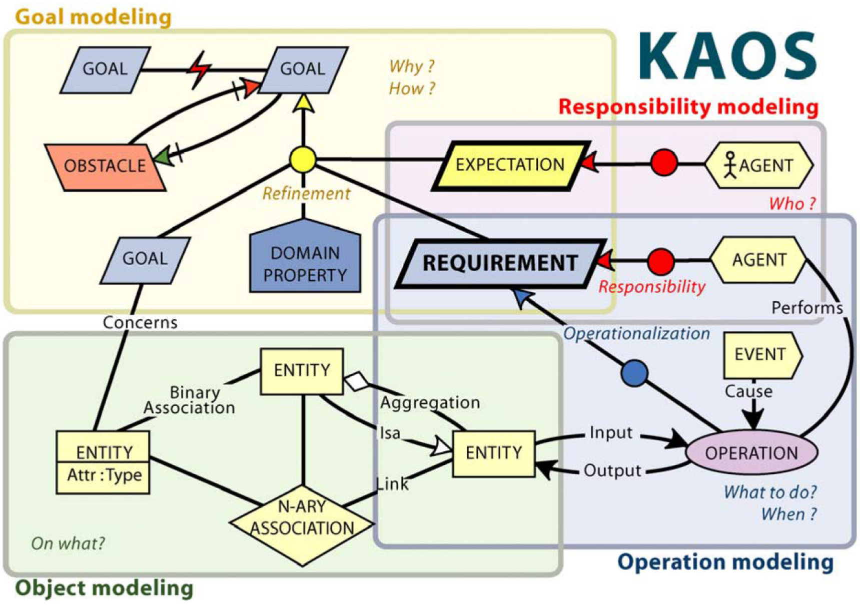


Figure 3.4: Overview over the KAOS meta-model.[1]

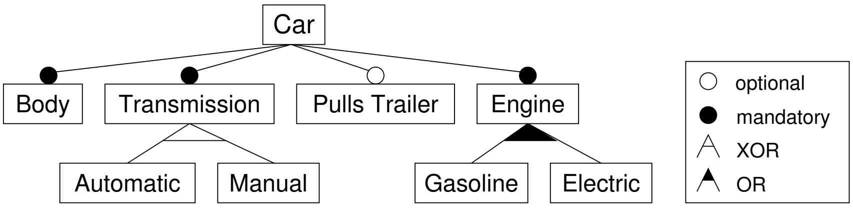


Figure 3.5: A feature model of a simple car [Apel and Kästner, 2009].

cies. Our main extension to the KAOS approach is to make function behavior and their dependencies explicit.

3.2.2 Feature Modeling

Making functions and their dependencies explicit has also been considered in the scope of product line engineering [Kang et al., 2002; Pohl et al., 2005]. In this context, most authors speak of features and feature interaction. Kang et al. [1990] introduce *Feature-Oriented Domain Analysis (FODA)* as a method to identify functional commonalities and differences of applications in a domain. FODA is best known for introducing feature models but it also proposes describing requirements of a software product line using a combination of an ER model and an integrated behavioral model parameterized by features. Functional features are organized in a hierarchical model that distinguishes between optional, mandatory, and alternative features and allows annotating features with a set of dependencies such as *excludes* or *requires*. Figure 3.5 shows an example of a feature model. FODA does not prescribe a particular behavioral modeling language, but Kang et al. [1990] give an example that uses statecharts. However, the statechart and ER models in the mentioned work are not

[1] http://www.objectiver.com/fileadmin/download/documents/KaosTutorial.pdf

interrelated. The focus of feature modeling approaches is on reasoning about configurations of features for different products rather than on specifying feature behavior or interaction. Extended feature modeling approaches, such as FORM [Kang et al., 1998] or FeatureRSEB [Griss et al., 1998], relate features with implementation artifacts to synthesize products from a selection of features.

Relation to our thesis In contrast to feature modeling approaches, we do not focus on variability aspects or product families but on a precise model of functions and especially their functional dependencies within a specification of a single multifunctional system. Our approach is able to model more complex function dependencies than just *excludes* or *requires*. Especially, we focus on describing *dynamic* function dependencies, i.e., behavioral dependencies, and not just *static* dependencies. Additionally, we associate each function with a behavior specification to be able to reason about the resulting functional specification of a system.

3.2.3 Feature-oriented Requirements Modeling

Feature-oriented requirements modeling is a requirements engineering paradigm that advocates (functional) features as first class entities of requirements specification and analysis. The approach we present in this thesis can be seen as a feature-oriented requirements modeling approach. The notion of a feature has also been considered in the context of the Jackson and Zave [1995] reference model. Classen et al. [2008] define the notion of feature as a subset of correlated elements from the RE reference model artifacts W (*world*), S (*specification*) and R (*requirements*). *"A feature is a triplet, $f = (R, W, S)$, where R represents the requirements the feature satisfies, W the assumptions the feature takes about its environment and S its specification."* Based on this definition, a set of features are said to interact, if

1. they satisfy their individual requirements in isolation,
2. they do not satisfy the conjunction of these requirements when put together,
3. removing any feature from the set of features results in a set of features that do not interact.

This definition relies on a very implicit notion of feature interactions. It is, for example, unclear how intended interactions between features are represented (e.g., in the requirements of one (or more) features?). Our approach relies on a much more explicit notion of feature interaction. In fact, functions in our approach interact if and only if an explicit interaction link, called a *mode channel*, is specified.

In the context of software product line (SPL) engineering, the feature-oriented requirements modeling language (FORML) was introduced by Shaker et al. [2012] for modeling the behavioral requirements of a software product line. A FORML model for an SPL is decomposed into two requirement views: A *world model*, which is an ontology of concepts that describes the problem world of the SPL; and a *behavior model*, which is an extended finite state machine model that describes the (behavioral) requirements for the SPL. The inputs of the behavior model are events and conditions about world phenomena and its outputs are actions over world phenomena. A behavior model is decomposed into feature modules, which describe the requirements

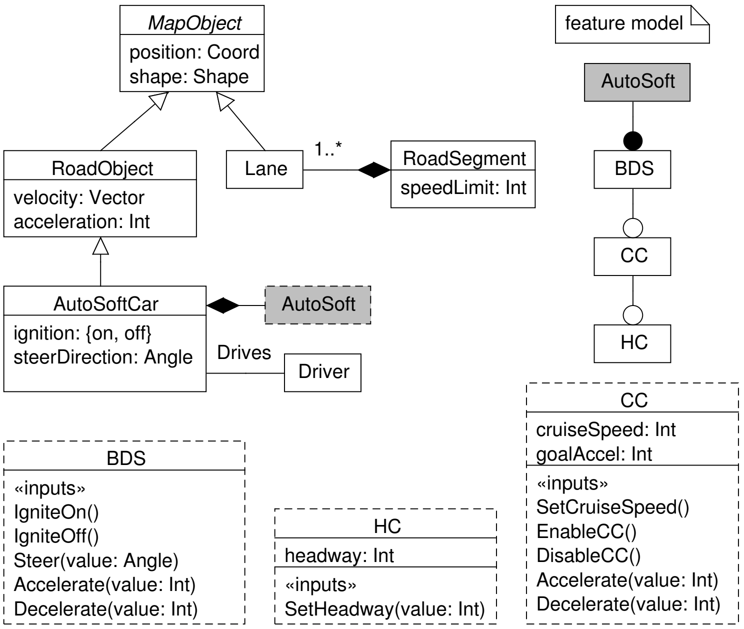


Figure 3.6: Example of a FORML world model [Shaker et al., 2012].

for features of the SPL. The world model contains a feature model defining the set of possible feature configurations in an SPL. The feature model is described in the original feature model notation FODA as introduced by Kang et al. [1990]. The feature model references *feature concepts* that describe the interactions of the SPL with its context implied by the feature. Figure 3.6 gives an example of a FORML world model for the SPL *AutoSoft*. The behavioral requirements of an SPL with a specific feature configuration are defined as the composition of the feature modules belonging to the features present in the feature configuration. A feature module can either add, remove, or replace behaviors in existing features. In all three cases, such requirements can be modeled as state machine *fragments* that extend existing feature modules at specified locations (cf. [Prehofer, 2013]).

Relation to our thesis The notion of a feature and its specification by a feature module in FORML is very close to the notion of a function used in this thesis. The function architecture presented in this thesis is similar to the behavior model in FORML but also supports to structure feature modules (called function specifications in our approach) in an architecture. In contrast to the FORML composition mechanism that is based on the addition of state machine fragments to an existing state machine, our approach is based on the parallel composition of interfaces. This has the advantage that the composition is defined by exactly one composition operator, whereas for the addition of a state machine fragment several cases have to be distinguished (e.g., transition priorities, transition overrides, strengthening and weakening clauses, additional regions or transitions). The introduction of a mode channel for a function can be compared to the addition of a weakening or strengthening clause fragment in FORML. The messages transmitted over a mode channel can be used to weaken or strengthen the conditions of a state machine transition.

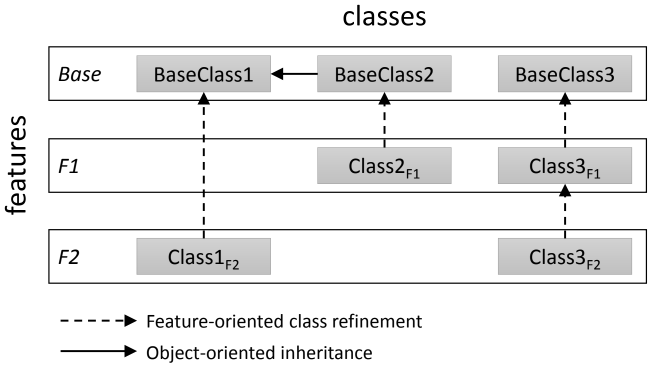


Figure 3.7: Feature-oriented class refinement (dashed arrows) and object-oriented inheritance (solid arrows) are concepts for reuse that are orthogonal to each other [Thüm et al., 2012].

3.2.4 Feature Implementation

Features have also been utilized to structure implementations on a programing language level. Following Dijkstra's *separation of concerns*, Prehofer [1997] proposed to structure code fragments with respect to features and calls that *feature-oriented programming (FOP)*. The implementation fragment that is associated with a feature is called a feature module [Batory et al., 2004]. Prehofer [1997] advocates a shift from the class hierarchy oriented view onto object-oriented programs to a feature repository oriented view. In classical object-oriented programs, specific functionality is added by inheriting from a base class and adding or overwriting methods of the base class. The result is a predefined hierarchy of classes exhibiting specific behavior. Prehofer [1997] generalizes this mechanism to independent features that can be added to any object. The concept of overwriting methods for changing an object's behavior in the presence of a specific feature is replaced by the definition of an additional module called a *lifter*. A lifter depends on two features and is a separate entity used for composition. The lifter contains the code that need to be changed in one feature implementation in the presence of another feature. In a program with n features, there is only a quadratic number of lifters but an exponential number of possible feature combinations [Prehofer, 1997]. Batory et al. [2004] describe the introduction of implementation modules for adding features to a program by a stepwise refinement process (see Figure 3.7). Later, Liu et al. [2006] formalize this idea of implementation modules that refine some base modules in the presence of a feature and call them *derivative modules*. All mentioned approaches rely on an object-oriented programming paradigm.

Relation to our thesis Especially in the approaches that originate from a focus on feature implementation, a feature is not necessarily declaratively complete and a developer may have to combine it with a base program or with other features. That means features are often increments in program functionality [Batory et al., 2004]. In the context of this thesis, a lifter/derivative can be seen as a specification increment to a function that has a mode dependency to another function. Prehofer's *"A* lifter *lifts an* object *to a* feature" [Prehofer, 1997] can be translated to "A *lifter* lifts a *function* to a *mode*" in the context of this thesis. However, the concept of lifters or any

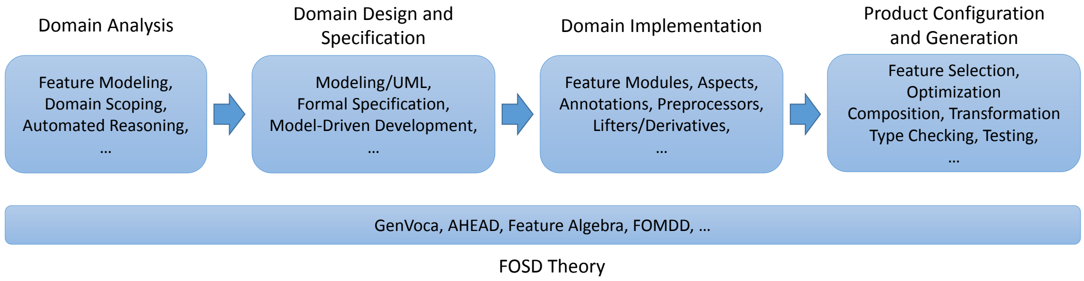

Domain Analysis
Feature Modeling, Domain Scoping, Automated Reasoning, …
Domain Design and Specification
Modeling/UML, Formal Specification, Model-Driven Development, …
Domain Implementation
Feature Modules, Aspects, Annotations, Preprocessors, Lifters/Derivatives, …
Product Configuration and Generation
Feature Selection, Optimization Composition, Transformation Type Checking, Testing, …
GenVoca, AHEAD, Feature Algebra, FOMDD, …
FOSD Theory

Figure 3.8: Phases of the FOSD process [Apel et al., 2013a].

equivalent does not explicitly exist in our approach since we leave it open how to realize a function in an implementation. Additionally, in our approach, functions of a system must not be composed in a specific order because they do not refine each other. Function specifications are self-contained (modular) and their interaction is solely coordinated via the mode model.

3.2.5 Feature-oriented Software Development

Feature-oriented software development (FOSD) is a paradigm for the construction, customization, and synthesis of large-scale software systems [Apel et al., 2013a; Apel and Kästner, 2009]. FOSD converged from work done in the fields of *feature modeling*, *feature interaction*, and *feature implementation* and aims at three properties: structure, reuse, and variation. FOSD integrates work from the mentioned fields into a feature-oriented specification and design process. The phases of this process and the corresponding methods and modeling languages are illustrated in Figure 3.8.

The FOSD process is, to some extent, backed up by different theories. Most theoretical work originates from the field of feature implementation and formalizes the composition of features. GenVoca [Batory et al., 2004] formalizes feature composition by a structural composition of code artifacts related to the features. In this theory, programs are described algebraically. Initial programs are constants and refinements are functions that add features to programs. A multifeatured (multifunctional) system is represented by an equation that is a named expression (e.g., $system = i(j(f))$ means $system$ has features i, j, and f). In GenVoca, a function represents both a feature and its implementation. A GenVoca constant represents a base program and is a set of classes. A GenVoca function is a set of classes and class refinements. A class refinement can introduce new data members, methods, and constructors to a target class, as well as extend or override existing methods and constructors of that class. The successor of GenVoca is called AHEAD [Batory et al., 2004] and generalizes the composition mechanism from the code level to arbitrary artifacts by representing programs as nested sets of equations. Base artifacts are constants and artifact refinements are functions. An artifact that results from a refinement chain is modeled as a series of functions (refinements) applied to a constant (base artifact). A similar algebraic formalization of feature-based program synthesis is described by Apel et al. [2010a]. In this foundation, a program p (which can itself be viewed as a feature) is composed of a series of features $p = f_n \bullet (f_{n-1} \bullet (\cdots \bullet (f_2 \bullet f_1)))$. The composition of features is defined by superimposition of their implementations [Apel and Lengauer, 2008].

Relation to our thesis The approach presented in this thesis focuses partly on the *domain analysis* but especially on the *domain design and specification* phase. However, we do not consider variability in either of the phases. The scope of this thesis does not include methods or techniques for the implementation of a multifunctional system. The focus is on its specification. However, we provide a mechanism to relate the specification with its abstract implementation, called its logical component architecture, based on the underlying modeling theory. The contributions made in this thesis provide guidance on how to work out and derive the different artifacts of the specific phases. On the other hand, the presented FOSD process could be applied as a process model to the artifact model presented in Chapter 5.

In all of the mentioned theories for FOSD, the order of feature composition is relevant since the composition mechanism relies on the idea that one feature extends another. In the modeling theory that is used in our thesis, the composition of functions is commutative due to the parallel composition operator [Broy, 2010b].

While the FOSD theories focus on the characterization of features by its implementations, it is an open issue, whether they can be extended to the other phases of the FOSD process. More generally, a unified theory that describes all structures and mechanisms used in the FOSD process is missing [Apel and Kästner, 2009]. The approach described in this thesis builds upon a unified modeling theory that is the basis for all employed modeling artifacts starting from requirements to design and implementation [Broy, 2010a,b].

3.2.6 Specification Notation Techniques

A specification model that comes very close to ours is the *Software Cost Reduction Method (SCR)* introduced by Heitmeyer [1998]; Heitmeyer et al. [2005, 1997]. Based on the four variable model of Parnas et al. [1994], SCR structures a specification into a set of tabular descriptions of input/output relations. Similar to our approach, there is also the notion of a mode, which can be used in the tabular descriptions to cut the conditions of specific specifications short. The modes themselves and their transitions are specified separately in mode transition tables. Similar to the table functions in SCR, mode transitions are triggered by inputs to the system (monitored variables) and their events. The SCR method has two drawbacks, which we try to overcome with our approach. First, we consider it a major limitation of SCR that mode transitions can only be triggered by conditions or events of the monitored variables. In our approach, mode transition can also be triggered by an output of a system function. This supports the definition of modes that reflect states of the controlling system itself. For example, a cruise control function in a vehicle may trigger a mode transition from *normal* to *follow-up*. This would be difficult to express as a function of monitored variables. Second, SCR provides no elaborated structuring or decomposition mechanism for specifications. In SCR, each dependent variable (i.e., controlled variable, term, or mode) is described by a separate table function. The composition of all table functions yields the specification of the entire system. Thus, the structure of a specification is determined by the dependent variables of the system. This makes it hard to modularize specifications, which is essential for multifunctional systems. Our approach structures the specification according to system functions, which is a powerful instrument to specify different parts of the specification independently.

Jackson and Zave [1998] introduced *Distributed Feature Composition (DFC)* as a modular, service-oriented architecture for applications in the telecommunication domain. DFC relies on the notion that a user service request can be composed of a set of smaller features, which are arranged in a *pipes-and-filters* architectural style. This architecture is especially designed for modeling interactions between different functions. The filters in this architecture are called *feature boxes* that can, if active, generate, absorb, or propagate signals as well processing or transmitting media streams [Zave et al., 2004]. Every service request is realized by a certain chain of feature boxes that are connected by internal calls. This chain, called a *usage*, describes the realization of a user function in a process-like, sequential manner. As a consequence, the order of feature composition in DFC plays an important role, in that the location of a feature in a composition determines what features it overrides (cf. [Apel et al., 2008; Shaker et al., 2012]). Feature interaction is an integral part of this architecture since it describes control and data flow between functions. In contrast, our approach aims at covering only functions that are directly observable by the environment and thus completely abstracts from realization details. These functions are provided concurrently by the system and function dependencies are only used to steer the behavior of a function. Therefore, the notion of a function used in this thesis is closer to the notion of a *usage* in DFC. Feature boxes in DFC are already considered as a logical description of a realization design (an architecture) in the context of this thesis.

UML [OMG, 2011] and SysML [OMG, 2012] are commonly used languages to describe software and system designs. These languages are often also used to describe the requirements of a system, for example by providing state charts, sequence charts or even block or class diagrams for expressing some kind of structure within the requirements. However, using UML or SysML has two major drawbacks in comparison to our specification technique. First, UML and SysML are designed to describe software and system architectures and not requirements or behavioral specifications. This becomes obvious when considering the weak expressiveness of use case diagrams or requirements diagrams, which are supposed to cover the requirements and the behavior specification in UML/SysML. A second drawback is that UML/SysML is often used in practice without any fixed semantics and even widely used tools allow using the languages in any (probably incorrect) way. Our notion of a function can roughly be related to the notion of a *use case* in UML [Booch and Rumbaugh, 1997]. Use Case Diagrams summarize and relate use cases to another, thus describing the family of functions of a multifunctional system. In contrast to Use Case Diagrams, our approach is more flexible and expressive, since Use Case Diagrams do not support hierarchical structuring, and relations can only describe *import* or *extend* relations between use cases. Additionally, behavior specifications for use cases in UML are rather informal.

In the context of feature implementation, specification techniques have been proposed that augment the elements of the implementation modules. Thüm et al. [2012] applied design by contract [Meyer, 1992] to feature-oriented programming [Batory et al., 2004; Prehofer, 1997] by introducing five approaches to define contracts for methods and their refinements in feature-oriented programming. They found that feature-oriented method refinement often requires the refinement of contracts such that the program specification depends on the actual selection of features. Instead of annotating implementation modules of a feature with specifications, Apel et al. [2010b] propose to first develop a *feature-oriented design* from a feature model, which defines valid feature combinations in a product family. The feature-oriented design serves as an analyzable specification for the feature implementation. The authors express this specification by an extension to the declarative specification language *Alloy* [Jackson, 2002] called *FeatureAlloy*. An Alloy specification model is composed of several different kinds of statements: *signatures*, which define the vocabulary of a model by creating new sets, *facts* which are constraints that are assumed to always hold, *predicates*, which are parameterized constraints that can be used to represent operations, *functions*, which are expressions that return results, and *assertions*, which are assumptions about the model. In FeatureAlloy, a feature is represented as a containment hierarchy, which encapsulates a collaboration of model elements (signatures, facts, etc.) that belong to a feature. Furthermore, FeatureAlloy supports the refinement of existing Alloy modules, signatures, facts, and so on by subsequent features without the need to modify existing model elements. The resulting specification can be checked for semantic dependencies and interactions between features. Both specification notations of Thüm et al. [2012] and Apel et al. [2010b] have a close connection to the implementation of a feature and target an object-oriented programming paradigm by specifying properties of objects or classes. In contrast, our formal system modeling theory specifies functions by system interface observations.

Relation to our thesis Despite the (sometimes only small) differences between the mentioned specification notation techniques and the specification technique used in this thesis, the main contribution of this thesis is not the introduction of a superior specification notation technique but rather the integration of a formal specification technique into a requirements engineering methodology. That means we provide a comprehensive description of model types and their relations that reach from general requirements to a precise functional system specification. This methodology aims at a specification that does not rely on any implementation paradigm but precisely defines a system's behavior.

3.3 Function Documentations

Function documentations summarize and document all relevant information for a function and thus serve as central artifacts in the specification, development, testing, and implementation of a function in an industrial context. In the following, we outline work related to the creation, structuring, and use of function documentations. The contribution of Chapter 6 builds up on and extends this state of the art.

3.3.1 Empirical Investigations of Function Documentations

Considering empirical data and evidences with respect to the use of function documentations in industry, we have found only little related work. Adam et al. [2009] report on lessons learned from several RE process improvement case studies with small and medium sized enterprises applying the Fraunhofer IESE ReqMan approach. In their paper, they experienced that 80% of their case study companies are challenged with writing requirements for the developers in an appropriate manner. They additionally define four *hot topics*, where the companies plan to make improvement, which are “elicit functional requirements”, “elicit non-functional requirements”, “review requirements”, and “document developer requirements”.

Relation to our thesis The results of the study on function documentations presented in Chapter 6 contribute to the exploration of the *hot topics* of Adam et al. [2009] by providing causes and explanations for the stated problems. More specifically, we identified two major problems in the current documentation of functions, namely the separation of the basic principle of operation from the technical implementation and the inadequate specification of logical conditions, states, and phases of a function. By addressing these challenges, we expect an improvement in the challenges faced in the study of Adam et al. [2009].

3.3.2 Structuring Function Documentations

Gross and Doerr [2012] describe the vision and advantages of view-based requirements documentations. They argue that in order to create high-quality requirements documentations that fit the specific demands of successive document stakeholders, the research community needs to better understand the particular information needs of downstream development roles.

Waldmann and Jones [2009] report on a beneficial implementation of a feature-oriented requirements engineering approach in the context of a Swiss hearing aid manufacturer. They chose to segment requirements by features, with each feature described in a separate, independently reviewed document, which typically contained multiple use cases, business rules, and nonfunctional requirements. As lessons learned from the process, they claim that a requirements meta-model is essential to consistently maintain document content. They additionally report on competing needs of developers for detailed design input information, and business stakeholders to limit the amount of time needed to review these documents. In this context, a strict separation of problem-space information from solution-space information proved unsuccessful because reviewers were not willing to spend time reviewing problem-space information alone. This corresponds to our experiences when discussing possible levels of abstraction with different roles.

Ferrari et al. [2013] state that the quality of a natural language requirements document strongly depends on its structure. A proper document structure improves the understandability of the requirements, and eases the modifiability of the overall requirements specification. In their work, they provide an automatic approach to evaluate the *relatedness* of requirements in a document by means of clustering algorithms.

Relation to our thesis The contributions presented in Chapter 6 provide answers to the information needs mentioned by Gross and Doerr [2012]. Our study reveals that for downstream development roles, a high-level description of a function is important specifying the basic principle of operation and abstracting from implementation details. Our results additionally show that, depending on the task to fulfill, some information within the document is more relevant than others. This supports the idea of view-based documentations that is suggested by Gross and Doerr [2012]. Similar to the claims of Ferrari et al. [2013], our study also revealed that a proper document structuring was considered crucial for the quality of function documentations. Our study results suggest structuring a function documentation with respect to modes of a function.

3.3.3 Standards and Templates

Besides academic approaches, there exist a number of standards and templates for the documentation of functions.

The IEEE has released standard 29148 for software requirements specifications (SRS) and system requirements specifications (SyRS). The standard recommends specifying system modes or states. *"Some systems behave quite differently depending on the mode of operation. For example, a control system may have different sets of functions depending on its mode: training, normal, or emergency."* [ISO/IEC/IEEE, 2011b]. The SRS annex contains an alternative documentation structure for systems with modes.

Another well-known template for requirements documentation is the Volere Requirements Specification Template[2]. System states or modes are not explicitly covered in this template. Robertson and Robertson [1999] suggest to group functional requirements by use cases. *"The advantage achieved by doing so is that it becomes easy to discover related groups of requirements to test the completeness of the functionality."* Grouping requirements according to use cases corresponds to grouping them according to functions as performed in our approach. However, our study shows that solely applying this criterion is too coarse-grained and a structuring according to modes should be applied for each function/use case.

Relation to our thesis The function documentation structure provided by IEEE 29148 is similar to the one we describe in Chapter 6. However, the standard does not consider describing requirements on different levels of abstraction. Our study reveals that a high-level description and structuring of function requirements is demanded by the stakeholders of function documentations for multifunctional systems as well as their refinement to more technical and fine-grained requirements. In Chapter 6, we provide a function documentation template that integrates a structure according to states with a description on different levels of abstraction.

3.4 Elicitation of Mode Models

In the following, we list work related to the specification of systems based on states/ modes and approaches to elicit mode models. The contribution of Chapter 7 builds up on and extends this state of the art.

[2] `http://www.volere.co.uk/template.htm`

3.4.1 State-based Specifications

The idea of modeling requirements by means of states is described in many approaches (e.g., [Broy, 2010b; Dietrich and Atlee, 2013; Heitmeyer et al., 1997]).

Broy [2010b] provides the formal basis for the methodology we present in this thesis. He proposes to capture the requirements for a multifunctional system as services and structure them in a service hierarchy.[3] He specifies the services based on a formal system model, in which a service is described as an input processing and output producing function. Functional dependencies between services are described by inter-service communication, which the author calls mode channels. The hypothesis behind this is that inter-service communication can always be described by a mode (e.g., "If the car is driving faster than 20 km/h, the video player should be switched off."). However, a systematic elicitation approach for these modes is missing.

The situation is similar for the Software Cost Reduction Method (SCR) [Heitmeyer et al., 1997]. In SCR, *mode classes*, whose values are called *modes*, are used as auxiliary variables for the concise specification of required system behavior. A systematic approach for eliciting these mode classes is also not given for SCR.

Dietrich and Atlee [2013] provide a generic structure for a function, which they call feature, into the three high-level states *Inactive*, *Active*, and *Failed*. In real examples, this basic structure is too general in most cases. Almost all states need to be defined in more detail to provide an expressive model of the general solution concept. During an *Active* state, for example, a function may run through a number of additional states or phases. Additionally, a function might have a number of different degraded behaviors depending on the failure detected. The elicitation approaches presented in Chapter 7 result in much more fine-grained models of system modes.

Relation to our thesis The major research deficit in all of the stated work is a missing systematic elicitation approach for modes of a system. This is critical because an incomplete, inconsistent, or imprecise mode model may invalidate the results of a state-based specification or analysis approach. Moreover, the stated specification techniques refer to modes only as auxiliary means to provide a concise specification. In our thesis, we will show that a mode model by itself is a valuable entity for understanding and specifying a system.

3.4.2 Requirements Formalization

Filipovikj et al. [2014] mention the use of high-level concepts, such as *shutdown* or *start-up*, in textual requirements as an impediment to their formalization. In their study, they report on the difficulties inherent to the process of transforming system requirements from their traditional written form into semi-formal notations by applying specification patterns [Dwyer et al., 1999]. They observed two problems related to the mentioned high-level concepts. First, none of the predefined *scopes* of the specification patterns capture the moment when the system is "in" some specific state (e.g., *start-up*), and secondly, such states belong to a higher abstraction level and

[3] In other publications, Broy uses the same terminology as we do in this thesis (i.e., function instead of service, and function architecture instead of service hierarchy).

are not properly specified, so their meaning is ambiguous. The authors conclude that such high-level concepts need to be disambiguated by an engineer. Post et al. [2012] performed a similar study. The examples of requirements they present as difficult/ impossible to formalize also contain references to states (e.g., "The drag torque and the activation torque depend on the operating state").

Relation to our thesis In the approach of this thesis, states or high-level concepts as described by Filipovikj et al. [2014] are modeled in a mode model. This mode model enables a precise definition of operational states of a system that can be referenced in requirements and specifications. Thus, the specification of a mode model may contribute to the formalization of requirements by providing high-level concepts with a precise meaning.

3.4.3 Safety Analysis Approaches

Well-known safety analyses also rely on the notion of system states. For example, Fault-Tree Analysis (FTA) is a top-down deductive safety analysis technique. It is primarily a means for analyzing causes of hazards [Leveson, 1995]. In an FTA, an analyst assumes a particular system state and a top event and then writes down the causal events related to the top event and the logical relations between them, using logic symbols. The leafs of the tree often correspond to states of components of the system. Thus, an FTA also results in a state model, however, only containing states causing a hazard. Failure Modes and Effect Analysis (FMEA) is an inductive analysis method, which allows studying the causes of components faults, their effects, and means to cope with these faults systematically. FMEA is used to assess the effects of each failure mode of a component on various functions of the system and to identify the failure modes significantly affecting dependability of the system. FMEA supplies information about failure modes of the individual components into the FTA. FTA and FMEA are often conducted together to complement each other [Troubitsyna, 2008].

Relation to our thesis FTA and FMEA do not provide any systematic approach for identifying failure modes [Leveson, 1995]. The elicitation approaches introduced in Chapter 7 may support an FMEA by specifying a set of possible system states.

Chapter 4

Empirical Study on Function Dependencies in Multifunctional Systems

In the problem statement of this thesis, we claimed that we need a comprehensive requirements engineering methodology that integrates and supports the specification of function dependencies. In this chapter, we present an empirical study conducted in industry that provides supporting evidence for this claim. In the study, we analyzed the component architecture of two productive automotive software systems with the aim to assess the extent and characteristics of dependencies between functions of the systems. Afterwards, we assessed whether the function developers were aware of these dependencies.

Our results show that within the component architecture, a large ratio of the analyzed vehicle functions depend on each other. These dependencies stretch all over the system and single functions have dependencies to half of all vehicle functions. We furthermore examined that the developers are not aware of a large number of these dependencies when they are modeled solely on an architectural level. Therefore, the developers mention the need for a more precise specification of function dependencies as part of functions specifications. These results challenge the current development methods and emphasize the need for a comprehensive requirements engineering methodology that integrates and supports the specification of function dependencies.

This chapter is partly based on previous publications [Vogelsang and Fuhrmann, 2013; Vogelsang et al., 2012].

4.1 Research Objective

The purpose of our two-phase, sequential mixed methods study is to obtain quantitative results from a sample and then follow up with a few individuals to explore those results in depth. In the first phase, a quantitative research question will address the extent, distribution, and characteristics of function dependencies in modern automotive software systems. In the second phase, qualitative interviews will be used to assess how aware the function developers are of the found function dependencies.

4.2 Study Design

In this section, we formulate the research questions, characterize the study object, and describe the data collection procedures. We then define the analysis procedures before we conclude with a description of how we ensure the validity.

4.2.1 Research Questions

The study examines the amount of function dependencies in automotive software systems and how dependencies are handled in the development process. We assess the awareness of existing function dependencies to justify the need for function specifications that consider function dependencies explicitly. We structured our study by four research questions.

RQ1: To what extent do dependencies between vehicle functions exist? We focus on dependencies in the sense that the behavior of a vehicle function is not solely dependent on its primary inputs but also on the state or data of another vehicle function.

RQ2: How are dependencies distributed over all vehicle functions? We are interested in the question whether dependencies are equally distributed over all vehicle functions and whether there are vehicle functions that are more central with respect to the dependencies.

RQ3: What categories of information characterize the dependencies? To develop and assess suitable modeling techniques and theories we have to understand the characteristics of the information that describes the dependencies. In this study, we aim at extracting different semantic concepts that stand behind these dependencies. That means we want to find out what the concepts and notions are that characterize a dependency.

RQ4: To what extent are developers aware of function dependencies? Developers of automotive systems are not necessarily aware of existing dependencies and interactions. We want to identify existing function dependencies that are unknown to developers as well as known dependencies that are not represented within the component architecture and assess them with regard to their plausibility.

Table 4.1: Overview of the study objects.

	MAN System	**BMW System**
Type:	Compact Truck	SUV
Number of vehicle functions:	55	94
Number of logical components:	462	325

4.2.2 Study Object

In our study, we analyzed two automotive software systems of modern vehicles and especially their component architectures. The first system we analyzed was a vehicle system from MAN Truck & Bus AG, which describes the entire software architecture of a compact truck. The system comprised 55 fully specified[1] vehicle functions that are realized by an overall of 462 logical components. Logical components may be used for the realization of more than one vehicle function. The second system was a vehicle system from the BMW Group. Within the component architecture, we focused on the driving dynamics and driver assistance domain. The system comprised 94 vehicle functions and 325 logical components. Table 4.1 summarizes the characteristics of the two examined systems. Please note, MAN Truck & Bus AG, BMW, and many other automotive companies denote logical components as *(virtual) functions*. Since the term *(virtual) function* can easily be mixed up with the term *vehicle function*, which describes a different concept, we use the term *logical component* instead (cf. [Broy, 2007]).

Logical components describe the realization/implementation of a vehicle function in a purely logical fashion, i.e., without any information about the hardware the system runs on. A network of logical components describes the steps that are necessary to transform the input data into the desired output data. An example for a system that consists of 3 vehicle functions, which are realized by a network of 6 logical components, is illustrated in Figure 4.1. The logical components are afterwards assigned to specific software components, which execute the behavior of the logical components. As a final step, these software components are deployed to a set of electronic computing units.

The relation between a vehicle function and a logical component in the context of this study is the following: A vehicle function is realized by a set of logical components that are arranged in a dataflow network. A logical component can contribute to the realization of a set of vehicle functions. Thus, there is an $n : m$ relation between vehicle functions and logical components. The set of all logical components and their connections form a logical component architecture of the entire system (see Section 2.2.2). The vehicle functions crosscut this architecture by the set of logical components that contribute to their realization (see Figure 4.1).

[1] The overall vehicle system comprises 142 vehicle functions. However, at the time of the analysis the system was not yet fully specified. Incompletely specified vehicle functions were excluded from the study.

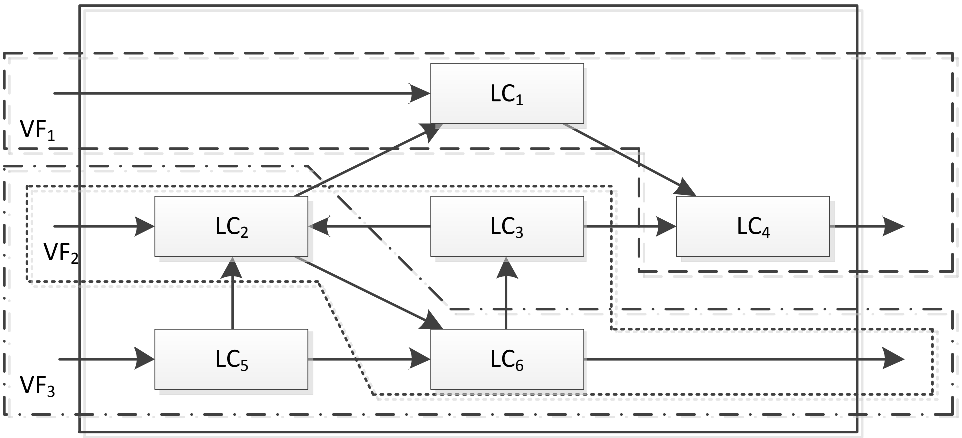


Figure 4.1: The logical components (rectangles) are connected by data channels (black arrows) and form a logical component architecture of the system (outer rectangle). Vehicle functions crosscut this architecture by the set of logical components that contribute to their realization (dashed forms).

4.2.3 Data Collection Procedures

For the reliable acquisition of data, we need a precise definition of what we consider as a dependency between vehicle functions. Our initial informal definition states that a vehicle function VF_1 depends on another vehicle function VF_2 if its behavior is influenced not only by its primary inputs but also by the state or data of VF_2. Therefore, the vehicle functions have to communicate with each other. In our study, we distinguish between two different ways of communication between vehicle functions.

1. A logical component that is part of one vehicle function has a communication channel to a logical component that is part of another vehicle function (see Figure 4.2).
2. A logical component is part of two vehicle functions, i.e., the vehicle functions share a logical component (see Figure 4.3).

A dependency between vehicle functions in the second case, however, cannot be ensured definitely without further knowledge about the behavior of the logical components. In the example of Figure 4.3, the influence of the data transmitted over the channel $LF_1 \rightarrow LF_3$ on the data transmitted over the channel $LF_3 \rightarrow LF_5$ can only be assessed with further knowledge about the behavior of LF_3. As we had no information about the precise behavior of logical components in the context of our study, we focused on dependencies of the first type, where a logical component of one vehicle function has a communication channel to a logical component of another vehicle function. This, at least gives a lower bound for the dependencies. Our analyses eventually showed that this restriction does not adulterate the results excessively. Considering all dependencies of the second type as actually being dependencies, the number of the dependencies in the system just increased by around 10%.

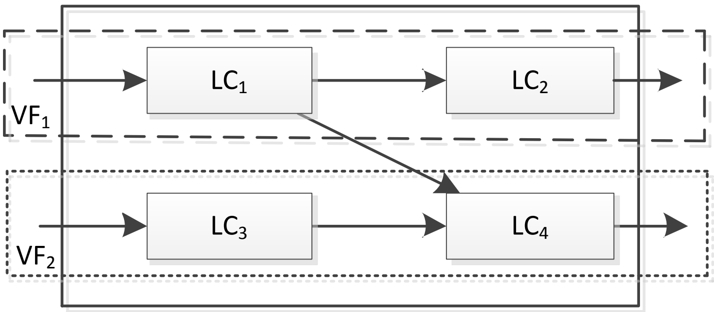


Figure 4.2: The vehicle function VF_2 depends on the vehicle function VF_1, since the logical component LC_1 (part of VF_1) sends values to the logical component LC_4 (part of VF_2). Thus, the behavior of VF_2 depends on data of VF_1.

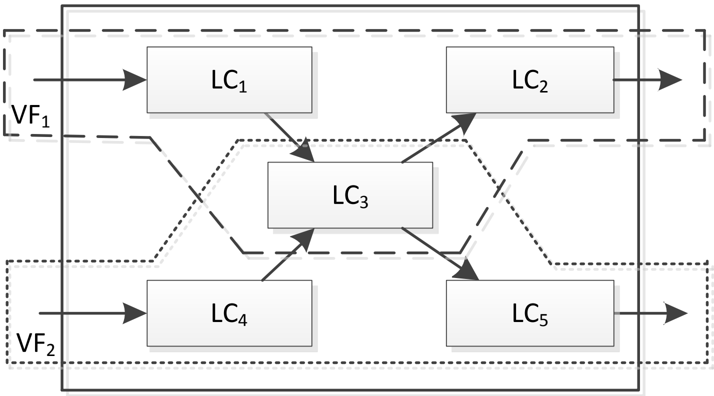


Figure 4.3: Vehicle functions VF_1 and VF_2 share the logical component LC_3. This might indicate a dependency between the vehicle functions. However, this cannot be verified without further knowledge about the behavior of LC_3.

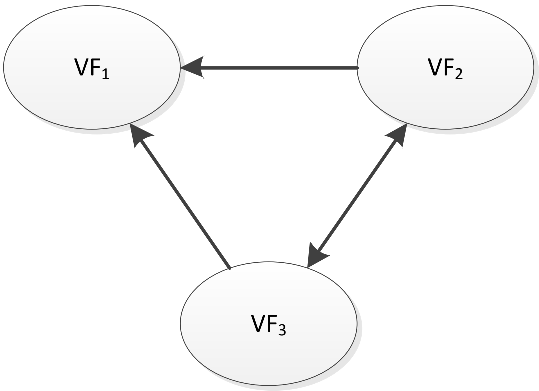


Figure 4.4: The vehicle function graph extracted from the logical component architecture of Figure 4.1.

Based on this definition of a dependency between vehicle functions, we extracted a vehicle function graph from the logical component architecture, where each node is a vehicle function and a directed edge indicates a dependency between two vehicle functions. The resulting vehicle function graph for the example of Figure 4.1 is illustrated in Figure 4.4. In our study, we extracted the vehicle function graph by means of a simple tool, written in Java. The tool parses an exported data set from the company's data backbone containing a list of vehicle functions associated with a set of logical components. The tool transforms the data into a graph structure, extracts the function dependencies according to the definition given in this section, and finally outputs a `.csv` file with the found dependencies. The extraction was performed fully automated and the complexity of the algorithm is quadratic in the number of vehicle functions and logical components. For the observed system, the extraction took around 3 seconds on a standard laptop.

The second part of our study is based on four interviews with function experts from the BMW Group, who are involved in the design of the logical component architecture. For RQ 4, we confronted the experts with a sample of function dependencies from their area of responsibility found by our analysis. We let the experts classify these dependencies into the following categories:

- **plausible/implausible:** A dependency is considered as *plausible* if the expert finds a functional or physical explanation for this dependency. If the expert has no functional or physical explanation for this dependency, it is considered as *implausible*.
- **known/unknown:** A dependency is considered as *known* if the expert was aware of this dependency prior to the interview. If the expert was not aware of this dependency prior to the interview, it is considered as *unknown*.

Overall, we discussed 100 function dependencies in depth.

4.2.4 Analysis Procedures

To answer the research questions, we collected the following measures from the study object:

For RQ 1, we analyzed the vehicle function graph to assess the ratio of vehicle functions that dependent on another vehicle function and to count the number of incoming and outgoing dependencies between vehicle functions. This gives an impression of the extent of function dependencies in realistic systems.

For RQ 2, we measured the dependency fan-in and fan-out as well as the PageRank [Brin and Page, 1998] for all vehicle functions of the vehicle function graph to see whether dependencies are distributed equally or if certain vehicle functions are more central than others are. Thus, we obtain information about the distribution of function dependencies in real automotive software systems.

For RQ 3, we classified the function dependencies of the MAN system by the type of data that realize the dependencies. For this purpose, we examined the communication channels between two logical components that cause the corresponding vehicle functions to be dependent and classify the data that is transmitted over the channels. We group the data channels by their data type into three categories:

Value: A data type that expresses a value on a continuous scale (a physical unit in most cases)

Enum: A data type that expresses a value on a discrete finite scale with more than two values

Bool: A data type that expresses a value on a scale with two possible values

This categorization should provide additional information about the semantic concepts that characterize the dependencies. We expect to get a better understanding of the nature of such dependencies from this categorization.

For RQ 4, we counted the number and ratio of function dependencies in the BMW system for each combination of category values, leading to a 2x2 matrix with the two categories as dimensions. We especially investigated the ratio of *plausible* function dependencies as an indicator for the validity of our quantitative study and the ratio of *known* function dependencies as an indicator for the awareness of function dependencies in general.

4.2.5 Validity Procedures

To ensure internal validity for the MAN system, we analyzed the system under investigation at a stage where it was already subject to an architectural review. This way, design flaws and misconceptions within the analyzed model should be reduced. Additionally, we presented and discussed the results with the developers and the responsible persons at MAN, who ensured that our results are valid and reasonable.

For the BMW system, we analyzed the system under investigation at a final stage of the development process where it was already subject to several architectural reviews and testing procedures. Therefore, errors and misconceptions in the logical component architecture can nearly be ruled out. However, our analyses show that vehicle functions might differ in the way they are modeled within the logical component architecture. To ensure validity we presented and discussed the results with function experts at the BMW Group, who assessed the found dependencies concerning their plausibility.

Table 4.2: Extent of dependencies in the vehicle function graph.

	MAN System ($n = 55 \mathrel{\widehat{=}} 100\%$)		**BMW System** ($n = 94 \mathrel{\widehat{=}} 100\%$)	
Vehicle functions. . .	**Number**	**Ratio**	**Number**	**Ratio**
with incoming dependencies	36	65.5%	81	86.2%
with outgoing dependencies	29	52.7%	72	76.6%
with incoming and outgoing dependencies	27	49.1%	68	72.3%
without dependencies	17	31.0%	9	9.6%

The evaluation of RQ 4 is based on interviews with function experts from the BMW Group. In order to get representative results from the interview partners we selected one expert from each area within the domain of driving dynamics and driver assistance. These areas are lateral, longitudinal, and vertical dynamics as well as driver assistance features. The experts were responsible for a number of 12–46 vehicle functions.

4.3 Study Results

In this section, the results of the study are presented. They are structured according to the defined research questions.

4.3.1 Extent of Dependencies (RQ 1)

Table 4.2 summarizes the results of the analysis on the extent of dependencies in the two analyzed systems. Analyzing the vehicle function graph of the MAN system, we found 133 dependencies between the 55 vehicle functions. 17 out of the 55 vehicle functions were completely independent from any other vehicle function and did not have any influence on other vehicle functions. 36 vehicle functions were dependent on another vehicle function (i.e., they had incoming dependencies) and 29 vehicle functions had an influence on another vehicle function (i.e., they had outgoing dependencies). 135 different data channels caused the dependencies.

Analyzing the vehicle function graph of the BMW system, we found 1,451 dependencies between the 94 vehicle functions. Only 9 out of the 94 vehicle functions were completely independent from any other vehicle function. 81 vehicle functions were dependent on another vehicle function and 72 vehicle functions had an influence on another vehicle function. 234 different data channels caused the dependencies.

Table 4.3: Distribution of dependencies in the vehicle function graph.

	MAN System (n=55)			BMW System (n=94)		
	Incoming	**Outgoing**	**PageRank**	**Incoming**	**Outgoing**	**PageRank**
Maximum	10	23	6.47%	48	53	5.81%
Median	1	1	1.26%	3	11	0.72%
Minimum	0	0	0.72%	0	0	0.28%

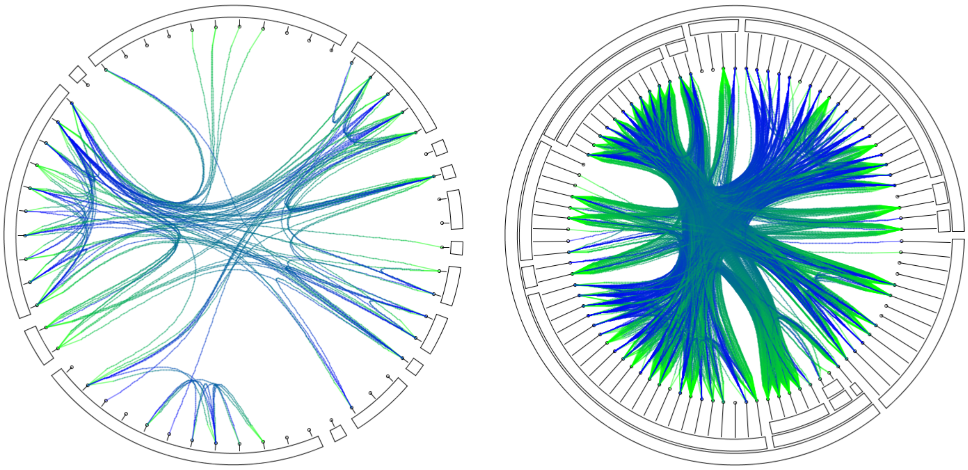


Figure 4.5: Vehicle Functions and dependencies of the MAN system (left) and the BMW system (right) visualized as an Edge Bundle View. The outer ring represents the hierarchy of vehicle functions. Each dot on the inside of the outer ring is an atomic vehicle function. The lines indicate a dependency between two functions.

4.3.2 Distribution of Dependencies (RQ 2)

The extent of the dependencies shows that dependencies between vehicle functions are distributed all over the system. However, some vehicle functions are more central in the sense that they have a number of dependencies to other vehicle functions. Table 4.3 illustrates this result. The table shows that for the MAN system vehicle functions have a maximum of 23 other vehicle functions that they influence, whereas one vehicle function depends on up to 10 other vehicle functions at maximum. For the BMW system, these numbers are even higher. A vehicle function depends on up to 48 other vehicle functions, whereas on the other side vehicle functions have a maximum of 53 other vehicle functions that they influence, which accounts for 56% of the vehicle functions. Most of the functions have at least 3 functions they depend on and have at least 11 functions they influence. The computation of the PageRank [Brin and Page, 1998] gives an idea about the “importance” of single vehicle functions and deviates by a factor of almost 9 for the MAN system and an even higher factor of almost 20 for the BMW system. This intermeshed structure of the functions becomes particularly visible when illustrating the dependencies in an *Edge Bundle View* [Holten, 2006] (see Figure 4.5).

Table 4.4: Categorization of dependency channels in the MAN system according to their data type.

	Overall		Unique	
	Number	**Ratio**	**Number**	**Ratio**
Value	122	26%	35	25.9%
Enum	314	66.8%	85	63%
Bool	34	7.2%	15	11.1%
Σ	470	100%	135	100%

Table 4.5: Plausibility and awareness of the analyzed function dependencies in the BMW system (n=100).

	known	**unknown**	Σ
plausible	41.0%	48.0%	89.0%
implausible	1.0%	10.0%	11.0%
Σ	42.0%	58.0%	100%

4.3.3 Categories of Dependencies (RQ 3)

As mentioned before, 135 different data types were involved in one or more dependencies between vehicle functions within the MAN system. To get a better idea of the characteristics of these data, we categorized, for the MAN system, the data channels that modeled a dependency according to their data type and grouped them into the three categories: *Value*, *Enum*, and *Bool*. The goal of this categorization is to get information about the kinds of data that is associated with the dependency in order to get a better understanding how to characterize the dependencies. Table 4.4 shows the results of this categorization. On the left side of the table, multiple occurrences of a data type are counted separately, whereas on the right side of the table every data type that contributes to any dependency is counted exactly once, independent from how often this data type contributes to any dependency. As the results show, this distinction has no influence on the distribution to the three categories. We additionally found that the channels of type *Enum* had 16 possible values at maximum. Our results show that more than 74% of the channels had a discrete and finite data type. One possibility to interpret these numbers is that a majority of the dependencies represent a propagation of state information or control commands from one vehicle function to another.

4.3.4 Awareness of Dependencies (RQ 4)

Table 4.5 summarizes the results of the expert interviews that we conducted for the BMW system in order to assess the plausibility and awareness of the analyzed function dependencies. The results indicate that our analysis produced reasonable results as only 11% of the examined function dependencies were considered as *implausible*, i.e., the dependencies were a result of our analysis but the experts considered them as not correct or at least they were not able to give account of them. Of the 100 function dependencies that we examined in the interviews, 42% were known to the experts and 58% were unknown. Most of the function dependencies that we examined were considered as unknown but plausible, i.e., the experts were not aware of the dependency between the functions but when examining the affected signals and logical components they found reasonable explanations for them. One examined dependency was considered as known and implausible as the expert was aware of it but had no explanation why this dependency exists.

4.4 Threats to Validity

Despite the applied validity procedures, the study design poses some further threats to the validity of the results. A threat to the internal validity is the fact that the analyzed model is already a realization/implementation of the vehicle functions. Dependencies might thus be a consequence of a design decision made by a developer and not an integral part of the system function itself. This effect can be seen as an instance of the *optional feature problem* [Kästner et al., 2009], which addresses that the implementation of features may be dependent, although the actual features are independent in the problem domain. Another threat pertains to the definition of dependency as given in this study. Besides the explicitly modeled dependencies that are in the focus of this study, there may also be dependencies between vehicle functions that occur when functions are implicitly connected by a feedback loop through the environment. Considering also such dependencies may additionally increase the number of detected function dependencies. The specification and detection of these dependencies would even require considering a system together with a precise specification of its context.

4.5 Discussion

The conclusions we draw point at a number of problems that occur in today's development of automotive software systems. Current development processes handle vehicle functions more or less as isolated units of functionality. To some extent, this has historical reasons as the automotive industry managed to make their different functionality as independent as possible such that vehicles could be developed and produced in a modular way. With the coming up of software-based functions in the vehicle this independence disappeared [Broy, 2006]. This is especially true for testing activities, where each vehicle function is tested separately. Behavior that arises from the complex interplay of vehicle functions due to its dependencies is not in the focus of these testing activities [Benz, 2010]. The extent and distribution of complex dependencies between vehicle functions also challenge the process and methods for specifying the requirements of a system. Each dependency between vehicle functions corresponds to a certain behavior that, in most cases, is observable by the user of a system and thus should reflect a functional user requirement. Therefore, a method for specifying system requirements should also account for dependencies between vehicle functions. Some vehicle functions have a large number of dependencies to other vehicle functions. They capture functionality that has an impact on many parts of the system, e.g., start-stop systems or energy management. Other vehicle functions have dependencies only to a certain group of vehicle functions.

Furthermore, in our interviews, function developers considered the knowledge about function dependencies as important. Reasoning about these dependencies on an architectural level is hard because function dependencies and design decisions are intermingled, which leads to the large number of unknown function dependencies as reported in the last section. An interesting point is that the reasons for function dependencies that were considered as implausible can also be related to architectural concerns. Logical components are architectural elements, which are subject to reuse and thus relate to a number of functions. Developers use and manipulate logical components without considering other vehicle functions that might also affect or be affected by this logical component. The emerging function dependencies were, in most cases, considered as implausible. Therefore, we argue that these dependencies need to be modeled precisely on the level of vehicle functions, still independent from any architectural design decisions (cf. [Broy, 2010b]).

Our results back up the challenges mentioned by Broy [2006]; Pretschner et al. [2007], where the authors state that *"functions [of a vehicle] do not stand alone, but exhibit a high dependency on each other, so that a vehicle becomes a complex system where all functions act together"*. The fact that most dependencies are represented by channels that have a discrete finite data type supports approaches, which model vehicle function dependencies by states of the vehicle (e.g., Broy [2010b]; Dietrich and Atlee [2013]).

4.6 Summary

The results of this study show that dependencies between vehicle functions pose a great challenge for the development of automotive software systems. Not only that almost every vehicle function depends on and/or influences another vehicle function, we have also seen that modeling the dependencies on an architectural level is insufficient for analyzing them, leading to a 50% chance that a developer is not aware of a specific dependency. In our study, this was particularly striking when the function dependencies arose from architectural decisions. Considering these conclusions, it becomes obvious that it is necessary to integrate the modeling of function dependencies on the level of vehicle functions into a comprehensive specification approach for such systems. In such an approach, functions have to be specified more precisely, for example by annotating them with inputs and outputs, and function dependencies have to be defined based on this notion of a vehicle functions (cf. [Broy, 2010b]). In such a structured specification approach, function dependencies are modeled independent from architectural decisions and thus the approach facilitates the modeling of feature interactions in requirements engineering.

Chapter 5

Integrating Functions and Modes into a Model-based RE Methodology

Model-based requirements engineering methodologies employ models to elicit, specify and analyze requirements. In this chapter, we introduce a model-based requirements engineering methodology that integrates the formal specification concepts for multifunctional systems described in Section 2.2 into model types and description techniques, which are created in an iterative process. By this contribution, we claim to enable an explicit and precise handling and specification of function dependencies during the different stages of requirements engineering and system development.

This chapter defines the RE methodology by introducing a set of model types and description techniques and their relations (Section 5.1), the role of these artifacts in a development process (Section 5.2), and two case studies, for which we instantiated the methodology and assesses its applicability, benefits and limitations (Sections 5.3 and 5.4). At the end of this chapter, we discuss the RE methodology by the lessons we learnt from the application in the case studies.

This chapter is partly based on previous publications [Böhm et al., 2014; Vogelsang et al., 2014].

5.1 Artifact Model

In Section 2.2, we described a system modeling theory that provides a formal characterization of semantic concepts, suitable for specifying multifunctional systems. The artifact model, as part of our RE methodology, relates these semantic concepts to model types that we use to describe them. The relations between the semantic concepts, as shown in Figure 2.1 of Chapter 2, must be reflected and expressed by the model types. We define these model types to systematically develop, capture, and analyze the semantic content of the system modeling theory.

Figure 5.1 gives an overview over the artifact model. The larger rounded boxes in the figure relate to the requirements engineering concepts of our semantic model (see Figure 2.1 in Chapter 2). Within the boxes, we depict the model types that we

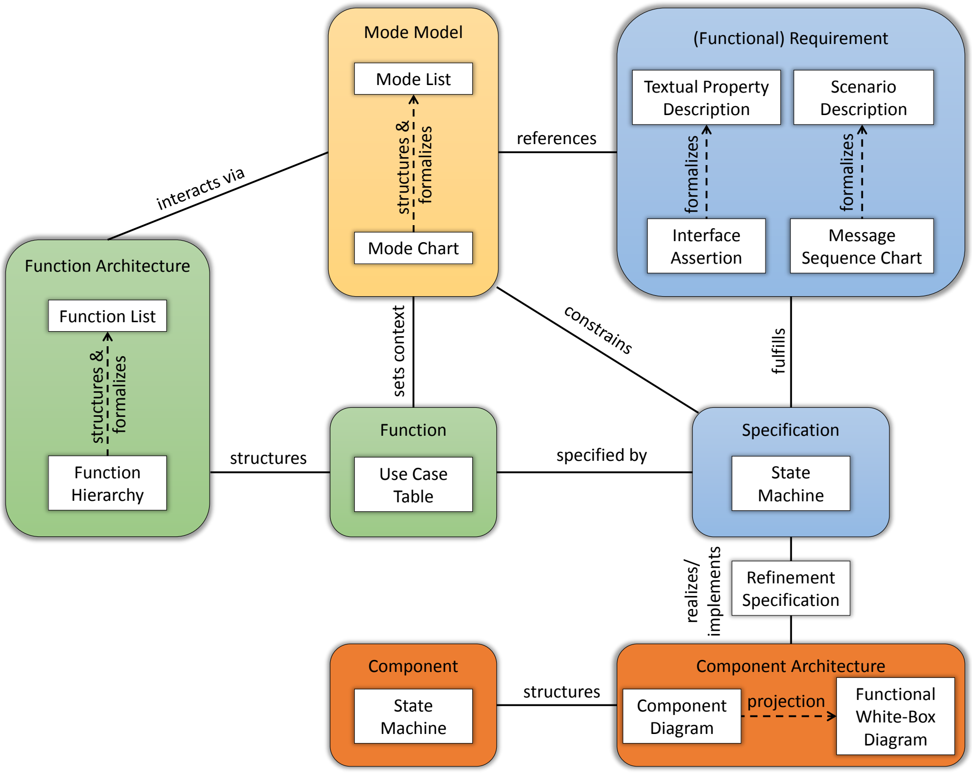


Figure 5.1: The artifact model of our model-based RE methodology associates the concepts of the semantic model with model types that we use to describe them.

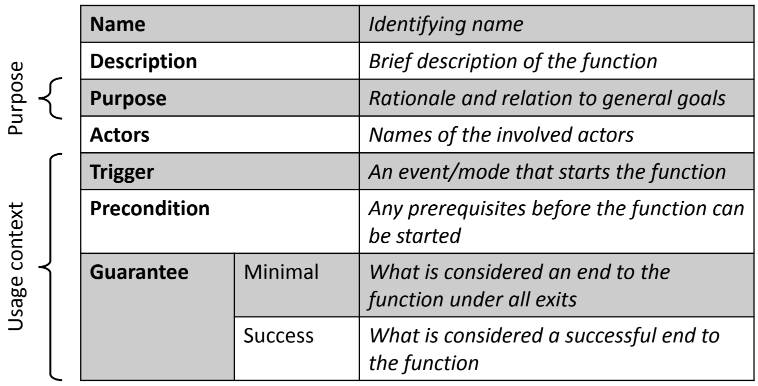
use to *describe* the respective content. For some semantic concepts, there are model types that describe the semantic concepts in a more formal or structured way than another model type (indicated by dashed arrows). Table 5.1 provides a short description of the model types used in the artifact model. In the remainder of this section, we describe the model types in detail and especially focus on how they reflect the relations between the semantic concepts.

Our artifact model provides a set of model types and relations between them that specify the functionality of a system with regard to different views and on different levels of formalization. In a concrete development project, this artifact model needs to be instantiated. In the instantiation process, a project manager decides, which model types should be created and in which order. This decision depends on the specifics of the development context (e.g., a certain level of formalization is needed for certification). In the case studies we present in Sections 5.3 and 5.4, we describe two examples of instantiating this artifact model in a concrete development scenario and discuss the implications.

In the following subsections, the constituents of the artifact model are described in detail. Section 5.1.1 describes the model types used to list and structure the functions of a system (function modeling), Section 5.1.2 describes the artifacts used to list and structure the operational modes of a system (mode modeling), Section 5.1.3 describes the artifacts used to specify desired behavior of the system (behavior modeling), and Section 5.1.4 describes the artifacts used to relate the resulting system specification to a logical description of the system implementation.

Table 5.1: Overview and short description of the model types used in the artifact model.

Activity	Semantic Concept	Model Type	Description
Function modeling (Section 5.1.1)	Function	Use case table	Informal description of a *function* including its name, purpose, actor(s), and usage context.
	Function architecture	Function list	A list of textual elements, each describing a *function* of the system by its name, a description, and its purpose.
		Function hierarchy	A hierarchical structure of function names with dependencies to each other representing the *function architecture*.
Mode modeling (Section 5.1.2)	Mode model	Mode list	A list of variables that represent the mode types of the *mode model*.
		Mode chart	A hierarchical structure of mode names with transitions describing valid mode configurations of the *mode model*.
Behavior modeling (Section 5.1.3)	(Functional) Requirement	Scenario description	Sequence of informally described interaction steps between the system and its environment.
		Message sequence chart	Formalization of a scenario expressed by a trace language based on message interchange.
		Textual property description	Textual statement about desired behavior, typically expressed in natural language.
		Interface assertion	Formalization of a textual property by a syntactic interface and an interface behavior expressed by a logic formula.
	Function specification	State machine	Specifies the behavior of a *function* by a state transition relation defining a syntactic interface and an interface behavior.
Architecture modeling (Section 5.1.4)	Component architecture	Component diagram	A hierarchical structure of component names connected via channels representing the *component architecture*.
		Functional white-box diagram	A projection of the *component architecture* showing only *components* relevant for the realization of one *function specification*.
	Component	State machine	Specifies a syntactic interface and an interface behavior of a *component* by a state transition relation.


Purpose
Usage context

Name		*Identifying name*
Description		*Brief description of the function*
Purpose		*Rationale and relation to general goals*
Actors		*Names of the involved actors*
Trigger		*An event/mode that starts the function*
Precondition		*Any prerequisites before the function can be started*
Guarantee	Minimal	*What is considered an end to the function under all exits*
	Success	*What is considered a successful end to the function*

Figure 5.2: Schematic representation of the model type *use case table* describing the semantic concept *function* with its purpose and usage context.

5.1.1 Function Modeling

As described in Section 2.2.2, a *function* is the intent to use a system in a specific usage context for a certain purpose. We represent these characteristics of a function by a *use case table*. A multifunctional system is a composition of such functions. We summarize the functions of a multifunctional system in a *function list* that is structured and formalized by a *function hierarchy*. Both model types are description techniques for representing the *function architecture* of a multifunctional system.

Use Case Table

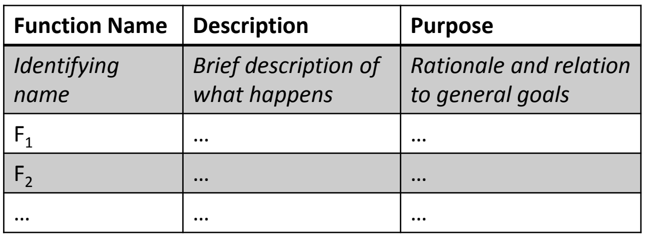
Use cases tables informally describe a *function* with its purpose and usage context from a user's perspective (cf. [Cockburn, 2001]). A use case table consists of a *name*, a *description*, a *purpose*, a set of *actors*, a *trigger*, a *precondition*, success and minimal *guarantees*. Figure 5.2 shows a schematic representation of a use case table. The entries of a use case table are described in natural language.

The *name* entry specifies the identifying name of the function that we also use as reference in other model types if we refer to that function.

The *purpose* entry specifies the intent of a user to use the system and originates from a user's goals. A more formal representation of the purpose of a function and its relation to intent and goals of stakeholders is not part of our methodology.

The *actor* entry specifies entities of the operational context of the system that are relevant for the use of the function. In general, this means that, while using the function it is necessary to get information from or provide information to the context entity. Context entities may be users or external systems. Usually, there is one main actor that has the intent to use the system for a certain purpose.

The *trigger*, *precondition*, success, and minimal *guarantee* entries of the use case table specify the *usage context* of a function. These entries characterize the intent of the main actor in more detail by providing information about situations, in which the main actor intents to use the system, or which situations the user intents to achieve.



Function Name	Description	Purpose
Identifying name	*Brief description of what happens*	*Rationale and relation to general goals*
F_1	…	…
F_2	…	…
…	…	…

Figure 5.3: Schematic representation of the model type *function list*.

The specification of the usage context is closely related to the mode model, which we describe later in more detail.

Function List

A *function architecture* as semantic concept is the composition of all functions of a multifunctional system. A *function list* represents the *function architecture* without information about the subfunction and dependency relation between functions (cf. Section 2.2.2). It provides a brief overview over the functions of a multifunctional system and can be used, for example, in an early stage of the development process.

A function list is an enumeration of distinct elements, each representing a function the system should provide. An item in this list refers to a function by its *name* and exhibits a short *description*, and the *purpose* of the function. Figure 5.3 shows a schematic representation of a function list.

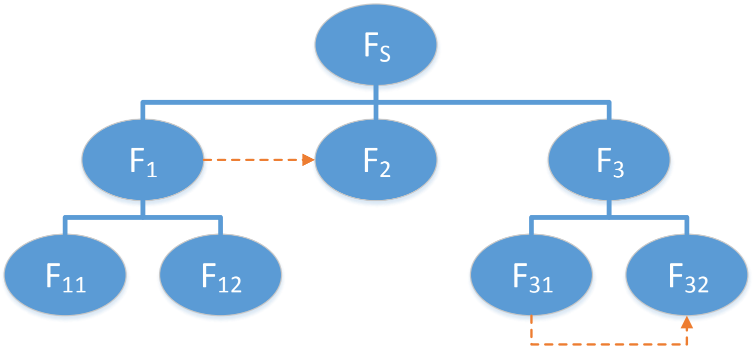
A function list is comparable to a list of *User Stories*, as applied in agile development methodologies, such as SCRUM [Schwaber and Beedle, 2001], where a user story captures desired functionality by templates (e.g., "As an <*actor*>, I want <*function*> so that <*goal*>").

The elements of the function list may reference a corresponding use case table by the identifying name of the function. In that case, the *description* and *purpose* entries are identical in both model types.

Function Hierarchy

A more complete and formal representation of the *function architecture* is given by a *function hierarchy*. In our methodology, a function hierarchy **structures** and **formalizes** the content of a function list. In a concrete development process, structuring and formalizing may be carried out in different stages of the process. Structuring the functions of a function list in a function hierarchy may be possible at an early stage, while the formalization may only be possible at a later stage when more information about the precise behavior of functions is present.

Structuring the function list: In a function hierarchy, the functions contained in the function list are structured in a hierarchy, where the root node represents the functionality of the whole system and the inner nodes represent a functional decomposition of this functionality into functions. Figure 5.4 shows a schematic represen-



Function List

Function Name	Description	Purpose
F$_1$	…	…
F$_{11}$	…	…
F$_{12}$	…	…
F$_2$	…	…
F$_3$	…	…
F$_{31}$	…	…
F$_{32}$	…	…

Figure 5.4: Schematic representation of the model type *function hierarchy* and the corresponding *function list*. Solid lines in the function hierarchy indicate the subfunction relation (e.g., F_1 is a subfunction of F_S) and dashed arrows indicate function dependencies (e.g., F_2 is influenced by F_1).

tation of a function hierarchy. In the figure, the system functionality (F_S) is broken down into subfunctions F_1, F_2, and F_3, of which F_1 and F_3 are further broken down to two subfunctions each. Each node in the function hierarchy (except for the root node) corresponds to one item of the function list.

Structuring the functions into a function hierarchy is a design decision that is based on a set of decomposition criteria, such as functional cohesion or organizational division. It is important to note that the decomposition of a function into subfunctions does not aim at providing a "solution" for this function (e.g., by a decomposition into sequential subfunctions). The decomposition rather aims at separating different "projections" of the system interface (e.g., by only considering a subset of the input/output channels). Penzenstadler [2011] gives a list of additional decomposition criteria. This list suggests, as one criterion, to summarize functions that a user of the system considers as associated (e.g., *speed limiter* and *speed control* as subfunctions of *driving assistance* in a vehicle). Still, this decomposition is a creative design process, which is not uniquely defined for a given system.

The figure also shows annotated dependencies between the functions F_1 and F_2, and between F_{31} and F_{32} (dashed arrows). These dependencies have a direction and indicate that the behavior of one function "influences" the behavior of the other function. Later, we will give this "influence" a precise meaning.

According to Penzenstadler [2011], function dependencies should also be considered when structuring the function list in a function hierarchy. On the one hand, function dependencies should be kept to a minimum; on the other hand, dependencies between functions that a user perceives as distinct should be made explicit (cf. *coupling* vs. *cohesion*).

In summary, the function hierarchy provides a structured view onto the function list by defining a subfunction relation and a dependency relation between the functions of the function list. Aside from that, the semantic concept of a function architecture as introduced in Section 2.2.2 also incorporates a formal interpretation of these relations, which is subject to the next paragraph.

Formalizing the function list: A function hierarchy can also be interpreted more formally, i.e., the structure that is established by the subfunction and the dependency relation between functions is equipped with semantics and thus becomes a precise specification.

For this purpose, we relate a function node in the function hierarchy to a *function specification*. Function specifications and their mode types, as introduced later in Section 5.1.3 in more detail, define an input/output behavior that is associated with the function in the function hierarchy. This input/output behavior is defined over a syntactic interface (cf. Section 2.2.1). The dependency relation between functions of a function hierarchy is interpreted as an exchange of messages via typed channels. The behavior of a function in the function hierarchy results from the parallel composition of all its subfunctions, as defined in Section 2.2.1. The behavior specification of an entire system results from the parallel composition of all its functions along the function hierarchy.

Figure 5.5 shows the function hierarchy from the previous example and adds schematic representations of function specifications for the functions F_1, F_2, and F_3. By this interpretation, we derive the system specification F_S by parallel composition:

$$F_S = F_1 \otimes F_2 \otimes F_3$$

Note, that from the point of view of the modeling theory, there is no difference between a leaf function, a function cluster (inner node), or the system specification (root node). The specification of each of these is a result of the parallel composition of specifications of their children in the function hierarchy.

5.1.2 Mode Modeling

A *mode model* as semantic concept is the model of operational states of a multifunctional system, which are called *modes*. We summarize the modes of a multifunctional system in a *mode list* that is structured and formalized in a *mode chart*. Both model types are description techniques for representing the *mode model* of a multifunctional system.

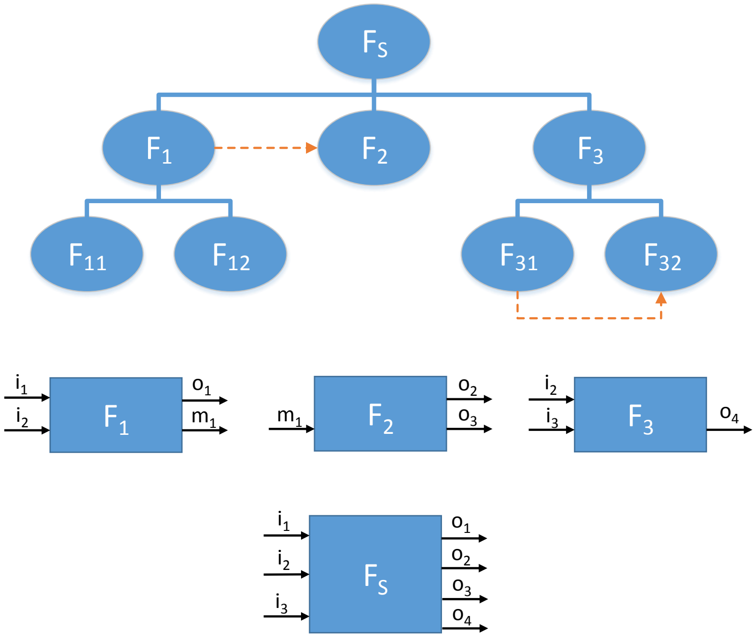
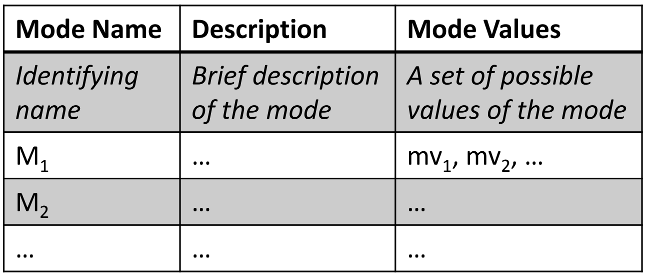


Figure 5.5: A *function hierarchy* is interpreted formally by relating the functions of the function hierarchy to formal *function specifications*. The formal behavior specification of the entire system results from the parallel composition of the function specifications related to the functions in the function hierarchy.



Mode Name	Description	Mode Values
Identifying name	*Brief description of the mode*	*A set of possible values of the mode*
M_1	...	mv_1, mv_2, ...
M_2	...	...
...	...	...

Figure 5.6: Schematic representation of the model type *mode list*.

Mode List

A mode list is an enumeration of variables that characterize the state space of the system and its operational context. An item in this list refers to a mode and is described by a *name*, a short *description*, and a set of *mode values* that characterizes possible values of the mode.

A *mode list* represents only the modes of a *mode model* that are also *mode types*, i.e., only modes with $kind = OR$ (see Section 2.2.2), without information about the submode relation and transitions between modes. Therefore, the mode list provides a brief overview over the modes of a multifunctional system that can be used to characterize the interaction between functions of a multifunctional system.

Figure 5.6 shows a schematic representation of a mode list. The mode list is motivated by the characteristics of the domain in which a system is applied and especially by its operational context.

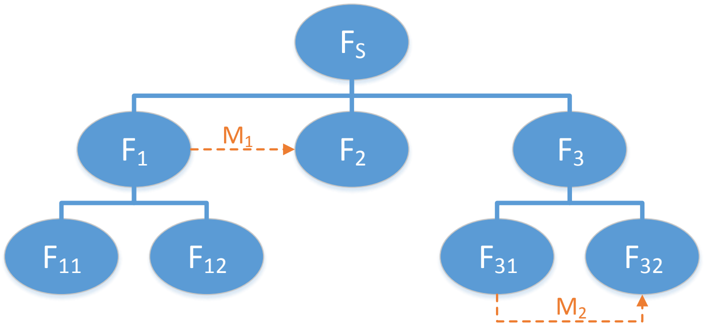


Figure 5.7: The modes contained in the *mode list* are referenced by the function dependencies in the *function hierarchy* (e.g., F_2 depends on the mode M_1 that is influence by F_1).

Modes play a special role for the description of dependencies between functions in the *function architecture*. A dependency between functions is characterized by one function that causes a change of a mode value (the *mode master*) and another that is influenced by this change (the *mode slave*) (cf. [Broy, 2010b]). Figure 5.7 shows how the mode list is referenced in the function hierarchy, which represents the function architecture by annotating a mode to each function dependency.

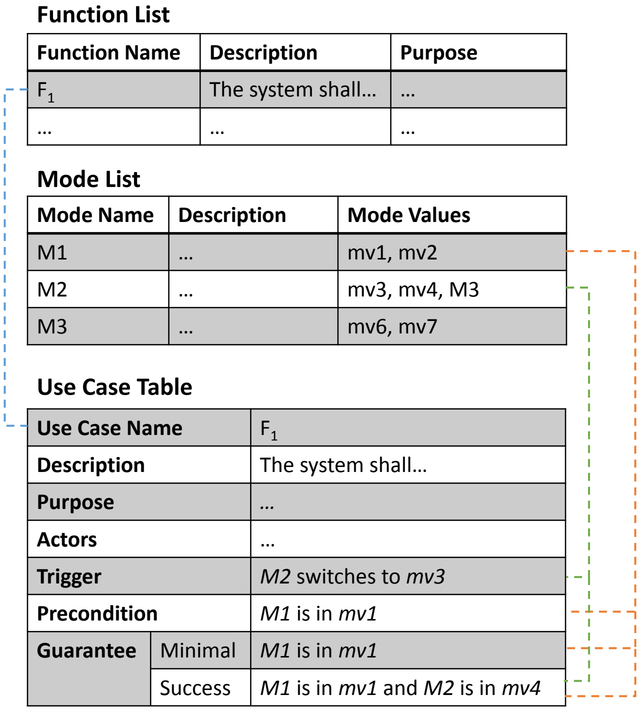
In the formal interpretation of the function hierarchy, as described in the previous section, the dependencies between functions are interpreted as behavior interactions between functions. These interactions are modeled by input/output channels between the functions. Since we annotate these dependencies with modes, we call the resulting channels *mode channels* (e.g., channel m_1 between the functions F_1 and F_2 in Figure 5.5 is a mode channel).

The mode model does not only play a role for the dependencies between functions of a function architecture but is also referenced in the description of the usage context of a function. The usage context of a function describes situations, in which an actor intents to use the system, or situations that the user intents to achieve. We model these situations by means of the mode model. Figure 5.8 exemplifies this by the relation between a use case table, the function list, and the mode list.

Mode Chart

A more complete and formal representation of the *mode model* is given by a *mode chart*. In our methodology, a mode chart **structures** and **formalizes** the content of a mode list.

In a mode chart, the modes of the mode list and their mode values are structured by defining the $children$ and $kind$ relation described in Section 2.2.2. We transfer a mode list to a mode chart by creating a mode with $kind = OR$ for each item of the mode list and defining the mode values as $children$. Additional *mode categories* that summarize a set of modes may be added to the mode chart as modes with $kind = AND$. Figure 5.9 shows a schematic representation of a mode chart. In the figure, the mode chart is structured into two parallel modes $M1$ and $M2$. $M2$ has three mode values, of which one mode value ($M3$) is a mode itself with mode values $mv6$ and



Function List

Function Name	Description	Purpose
F_1	The system shall...	...
...	...	...

Mode List

Mode Name	Description	Mode Values
M1	...	mv1, mv2
M2	...	mv3, mv4, M3
M3	...	mv6, mv7

Use Case Table

Use Case Name		F_1
Description		The system shall...
Purpose		...
Actors		...
Trigger		*M2* switches to *mv3*
Precondition		*M1* is in *mv1*
Guarantee	Minimal	*M1* is in *mv1*
	Success	*M1* is in *mv1* and *M2* is in *mv4*

Figure 5.8: The functions of a *function list* are detailed by *use case tables*. The usage context of a function is described in the use case table by referencing modes from the *mode list*.

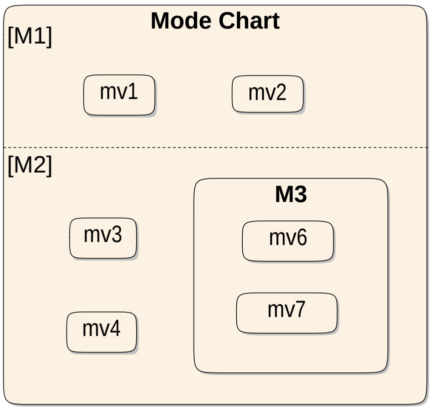
$mv7$. The mode model that is represented by the mode chart defines a set of valid configurations of mode values that characterize the operational state of the system.

As a second step, we extend the mode chart by adding transitions between the modes in the mode chart. These transitions define the possible sequences of mode configurations during the execution of a system. Additionally, an initial mode value for each mode needs to be provided.

Figure 5.10 shows a mode chart with transitions and initial mode values for each mode.

5.1.3 Behavior Modeling

In our approach, a *function* is associated with a behavior. Associating a function with a behavior changes the character of the term function. In the former sections, we characterized a function by its purpose and usage context. Associating a function with behavior specifies a concrete instantiation of using this function in terms of interactions between the system and its environment. Consider the function "Cruise Control" of a car. This function describes the intent to use the system "car" for holding the vehicle's speed on a desired level. An associated behavior may state that the driver first has to activate the "Cruise Control", which is confirmed by a system response, and after that, the driver can increase or decrease the desired speed based on the current speed at function activation. In general, many behaviors, i.e., scenarios of use, are imaginable for one function. In our semantic model, the behavior associated



Mode List

Mode Name	Description	Mode Values
M1	...	mv1, mv2
M2	...	mv3, mv4, M3
M3	...	mv6, mv7

Figure 5.9: The modes contained in the *mode list* are structured in a *mode chart*.

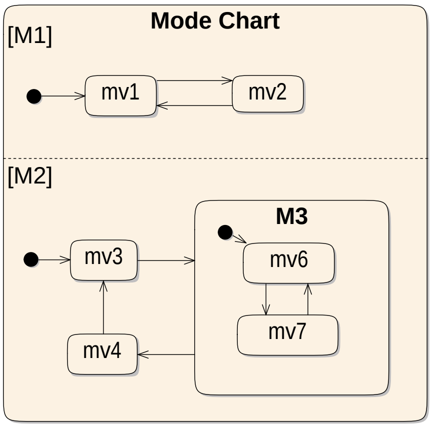
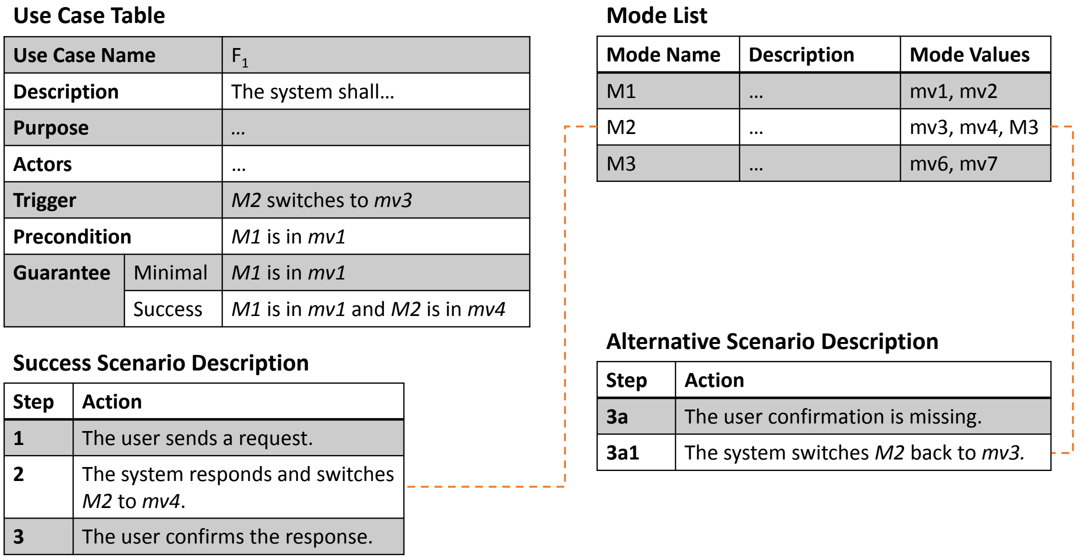


Figure 5.10: A *mode chart* with transitions and initial mode values for each mode.



Use Case Table

Use Case Name		F_1
Description		The system shall…
Purpose		…
Actors		…
Trigger		*M2* switches to *mv3*
Precondition		*M1* is in *mv1*
Guarantee	Minimal	*M1* is in *mv1*
	Success	*M1* is in *mv1* and *M2* is in *mv4*

Mode List

Mode Name	Description	Mode Values
M1	…	mv1, mv2
M2	…	mv3, mv4, M3
M3	…	mv6, mv7

Success Scenario Description

Step	Action
1	The user sends a request.
2	The system responds and switches *M2* to *mv4*.
3	The user confirms the response.

Alternative Scenario Description

Step	Action
3a	The user confirmation is missing.
3a1	The system switches *M2* back to *mv3*.

Figure 5.11: A *scenario description* supplements a *use case table* by a sequence of interaction steps that may reference modes from the *mode list*.

with a function is the *function specification* (see Section 2.2.2). In our methodology, we introduce I/O *state machines* as model type to represent the function specification.

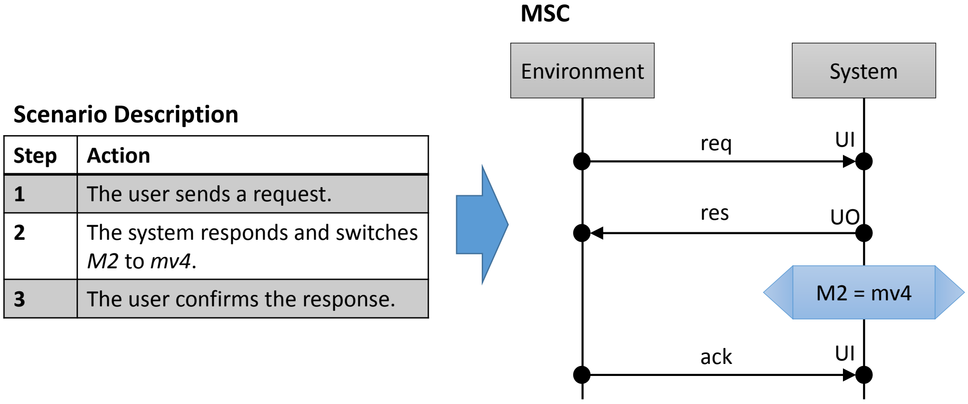
Specifying the behavior of a function is related to desired properties that the function should fulfill, i.e., to *(functional) requirements*. The idea is that a function specification fulfills a set of properties, which represent (functional) requirements. We introduce *scenario descriptions*, *message sequence charts*, *textual properties*, and *interface assertions* as model types to represent functional requirements of different character and on different levels of formalization.

Scenario Descriptions

The idea of describing functional requirements by specifying exemplary executions of the system, called scenarios, has attracted considerable attention in RE [Sutcliffe et al., 1998]. For example, Cockburn [2001], associates a use case with a set of scenarios in his RE methodology. In our methodology, a *scenario description* is a model type that defines a sequence of interaction steps between the system and its environment. The individual steps are still described in natural language.

Scenario descriptions can be used to advance from a purpose-based function description to a function specification by expressing exemplary executions that are associated with a function. The distinct steps in a scenario description are informal statements about information that is transmitted from a context entity to the system (i.e., input), from the system to a context entity (i.e., output), about a current operational state (mode condition), or about changes of an operational state (mode switch). The latter two types of steps, thereby, reference modes of the mode model.

Figure 5.11 shows a schematic representation of two scenario descriptions (success and alternative) with relations to the corresponding use case table and references to the mode list.



Scenario Description

Step	Action
1	The user sends a request.
2	The system responds and switches *M2* to *mv4*.
3	The user confirms the response.

Figure 5.12: Formalization of a *scenario description* by a *message sequence chart (MSC)*.

Message Sequence Charts (MSCs)

Our formal model characterizes *(functional) requirements* as properties defined over an interface (see Section 2.2.2). To be able to express scenarios as properties defined over an interface, we need a more formal representation of the steps given in a scenario description. A *message sequence chart (MSC)* represents a formalization of a scenario description by an interaction diagram providing a trace language based on message interchange [ITU-T, 2011]. Each scenario description is formalized by exactly one MSC. Figure 5.12 shows the formalization of a scenario description by an MSC. In the MSC, single steps of the corresponding scenario description are represented by the exchange of one or more messages between the system and its environment. Those messages, as opposed to the steps of a scenario description expressed in natural language, are already defined in terms of the underlying system modeling theory. That means a message is defined by a source port, a target port, and a value that is transmitted between the ports. A message either describes a stimulus (target port belongs to the system) or a reaction (source port belongs to the system). References to modes within the scenario description are expressed as conditions (represented by a hexagon).

An MSC that references a mode as a condition must be consistent with the mode model. That means, conditions that specify mode configurations that are not part of any valid configuration of the mode model are not allowed. Similarly, mode switches specified in an MSC must be reflected by transitions between valid mode configurations in the mode model.

Textual Properties Description

Textual property descriptions are another way of documenting desired behavior for a function. For some requirements, such as safety requirements, it might be easier to model them as statements to be valid in every situation of the system. This is contrary to the idea of a *scenario description*, where requirements are formulated with respect to a specific usage scenario. For some types of functions, it is even unintuitive to describe them as a set of scenarios (e.g., the speedometer function of a vehicle).

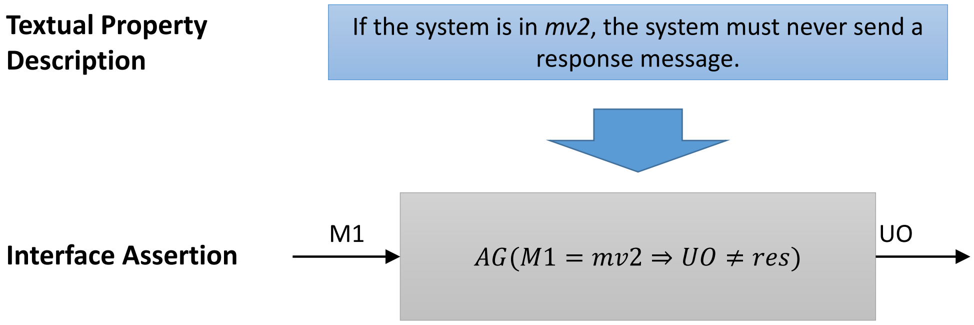

Textual Property Description
If the system is in *mv2*, the system must never send a response message.
Interface Assertion
M1
$AG(M1 = mv2 \Rightarrow UO \neq res)$
UO

Figure 5.13: Formalization of a *textual property description* by an *interface assertion* consisting of a syntactic interface description and an interface behavior expressed as CTL expression (temporal logic).

Textual property descriptions are natural language statements about desired behavior. In the course of the early requirements engineering phases, abstract and vague textual property descriptions need to be refined to properties that are more concrete. This helps explaining the origin and rationale of requirements to stakeholders. Furthermore, textual property descriptions may reference modes of the mode model. An example for a textual property description that references the mode value $mv2$ of mode $M1$ is "If the system is in *mv2*, the system must never send a response".

Interface Assertions

We introduce *interface assertions* as model type to formalize *textual property descriptions* by specifying a syntactic interface and an associated interface behavior expressed by temporal logic formulas. The syntactic interface specifies the stimuli and reactions that the textual property description refers to as input and output ports. It is important to note that these ports do not need to correspond to actual ports of the implemented system. Its purpose is rather to capture the events a textual property description speaks about. It is also important to note that it might not be possible to formalize a textual property description right from the beginning, especially if it is very vague (e.g., *the system shall be safe*). In such cases, the properties need to be concretized on the informal level, for example, by means of goal decomposition approaches (cf. [van Lamsweerde, 2001]). A formalization is possible as soon as the textual property description speaks about observable stimuli and reactions.

The referenced modes from the informal textual property description are formalized as inputs or outputs of the interface assertion. Figure 5.13 shows the formalization of a textual property description by specifying the stated events and conditions as inputs and outputs of the interface assertion. The behavior is captured by a CTL expression (temporal logic).[1] The textual property description states a stimulus that references the mode value $mv2$ of mode $M1$. This stimulus is modeled as an input of the interface assertion. As a reaction, the textual property description states to "never send a response message", which is modeled as an output of the interface assertion.

[1] In the case studies we will show that more user-friendly specification patterns can also be used to formalize the interface behavior.

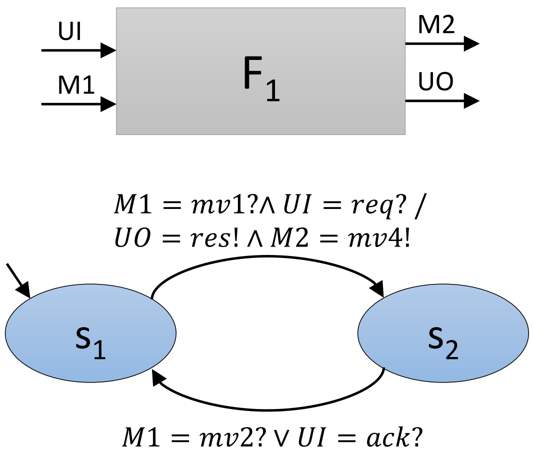


Figure 5.14: An I/O state machine describes a *function specification* with a syntactic interface (top) and an interface behavior (bottom) by a state transition relation.

An interface assertion is additionally constrained by the valid mode configurations specified in the mode model. An interface assertion is consistent with the mode model if its specified behavior only assumes and produces valid configurations of the mode model. If, for example, an interface assertion formalizes a property with $M2 = mv3 \wedge M3 = mv6$ as a condition, this interface assertion is not consistent with the mode model of Figure 5.10 because the mode model has no valid configuration that includes both $mv3$ and $mv6$. Interface assertions that also specify or assume mode switches must additionally conform to the mode transitions defined in the mode model. Similar constraints must hold if an interface assertion references a mode as an output.

Function Specifications by State Machines

A *function specification* associates a function with a behavior that includes and fulfills all desired *functional requirements* related to that function. We introduce I/O *state machines* as model type to represent the full formal characterization of a function specification as introduced in Section 2.2.2. That means, a state machine describes a syntactic interface and an associated interface behavior. While functional requirements and the model types we introduced to represent them only provide extracts of desired behavior, we introduce function specifications to progress towards a more complete functional system specification. Figure 5.14 shows a schematic representation of a function specification that is related to the interface assertion of Figure 5.13 and the MSC of Figure 5.12.

A function specification defines an interface behavior that needs to *fulfill* all associated interface assertions and MSCs. For an associated interface assertion this means that there is no execution of the function specification behavior that violates the logic property stated by the interface assertion. In other words, the state machine specification is a model for the logic formula. Fulfilling an associated MSC means that the MSC specifies a valid execution trace of the function specification. To match the execution trace defined by an MSC with the function specification, ingoing and outgoing messages of the system role in the MSC are mapped to input and output messages of the function specification. Conditions, expressed in the MSC by a hexagon, reference modes and are mapped to input or output ports of the function specification depending on whether they are conditions of the environment or the system. The MSC in Figure 5.12, for example, induces a function specification interface with one input port (*UI*) and two output ports (*UO* and *M2*). Additionally, the function specification must specify behavior that is consistent with the mode model, i.e., only relies on and produces sequences of valid mode configurations. Figure 5.15 provides an overview over the behavior modeling artifacts and their relations.

Function specifications are also used to interpret a function hierarchy formally. As described in Section 5.1.1, a function hierarchy is a structure of functions from the function list with additional dependencies between them. Function specifications provide the possibility to interpret this structure formally by associating a function specification to each function of the function hierarchy. The dependency relation in the function hierarchy is interpreted as communication between functions via mode channels. The subfunction relation in the function hierarchy is interpreted as composition of function specifications (composition as introduced in Section 2.2.1).

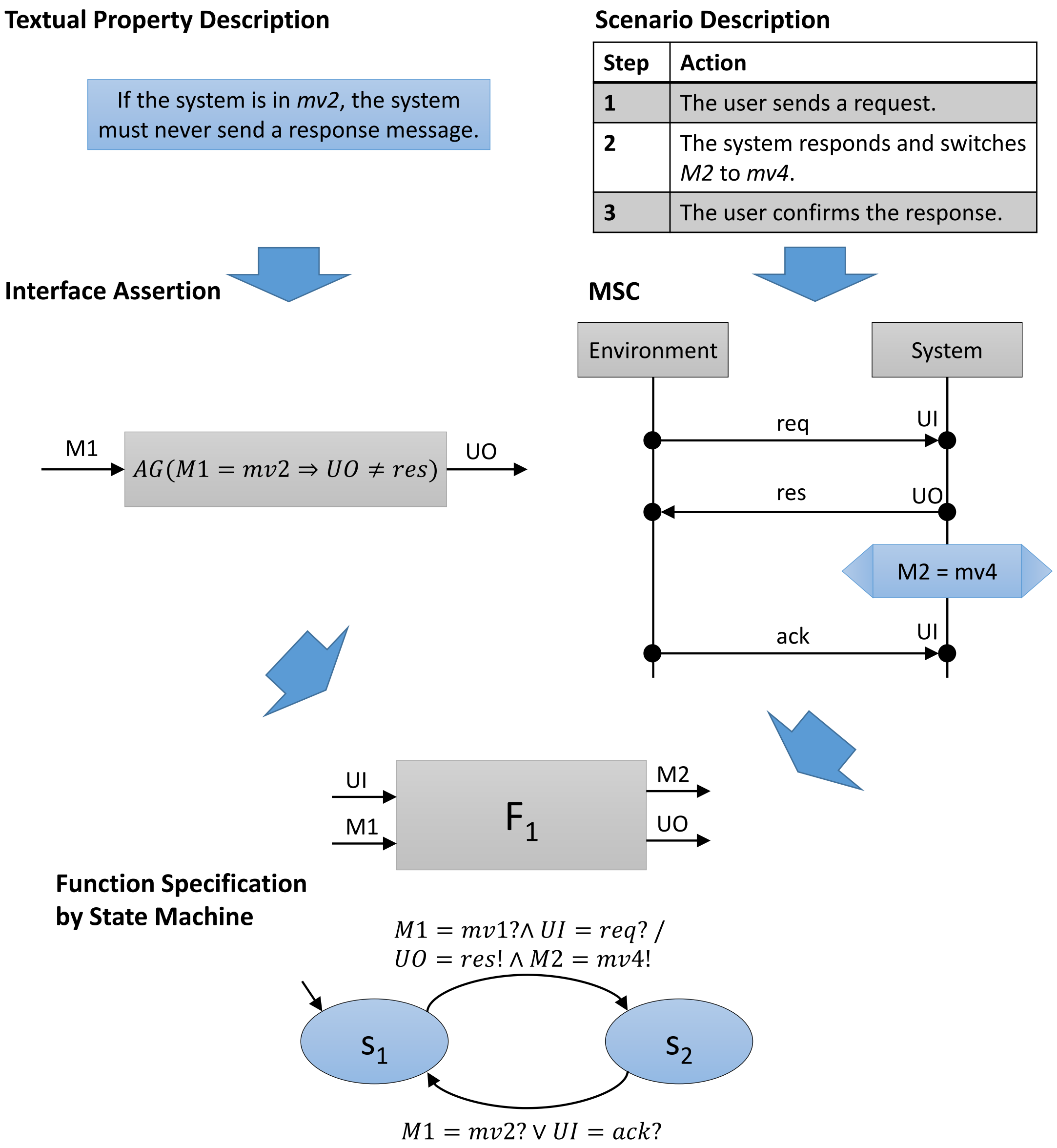
5.1.4 Relating RE Artifacts to System Architecture

The function architecture and the associated function specifications serve as system specification. The functionality described in the specification must be realized by the implemented system. An implemented system is a solution for the system specification. This solution contains an architecture that describes the design of the implementation. We differentiate between a component architecture (see Section 2.2.2), which describes the internal logical structure of a solution and a technical architecture (see Section 2.2.2), which describes the technical structure of the solution in terms of the target execution platform. In the following, we show how the RE model types are related the component architecture. The connection between the component architecture and its deployment to a technical execution platform is not in the scope of this thesis.[2]

We represent the component architecture of a system by a *component diagram*. Afterwards, we connect the *function specifications* to the *component architecture* by a *refinement specification*, which is introduced later.

Component Diagram We model the component architecture of a system by a hierarchy of communicating components. Components have typed ports, which describe the syntactic interface of the component. Ports can be connected by channels to specify a message transmission between components. We represent a component architecture by a *component diagram*. Figure 5.16 shows a schematic representation of a component diagram. In the figure, the system architecture consists of components C_1, C_2, and C_3. For each component in the component diagram, we need to provide a *component specification* that, similar to a *function specification*, describes the interface behavior of a component by an executable specification technique, such as state machines, specification tables, or code specifications.

[2] Details on the relation between component and technical architecture are, for example, given by Kugele [2012] and Voss and Schätz [2013].


Textual Property Description
If the system is in *mv2*, the system must never send a response message.
Scenario Description
Interface Assertion
M1
$AG(M1 = mv2 \Rightarrow UO \neq res)$
UO
MSC
Environment
System
req
UI
res
UO
M2 = mv4
ack
UI
UI
M1
F_1
M2
UO
Function Specification by State Machine
$M1 = mv1? \wedge UI = req?$ /
$UO = res! \wedge M2 = mv4!$
S_1
S_2
$M1 = mv2? \vee UI = ack?$

Step	Action
1	The user sends a request.
2	The system responds and switches *M2* to *mv4*.
3	The user confirms the response.

Figure 5.15: Function specifications incorporate the behavior defined by textual property descriptions, scenario descriptions, and their formalizations.

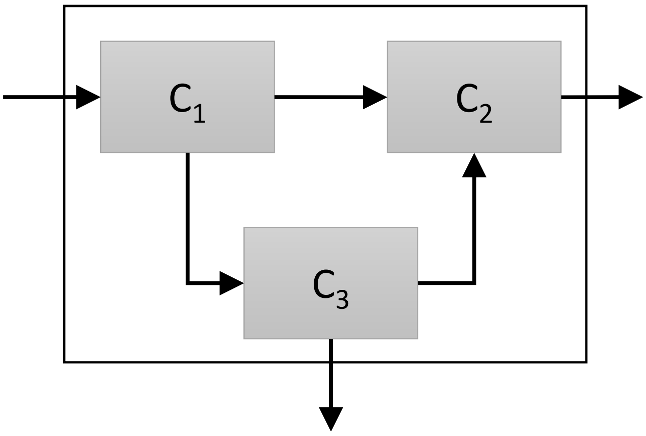


Figure 5.16: Schematic representation of the model type *component diagram* representing the *component architecture*.

Side Note: On the Difference between Function and Component

A fact that is sometimes confusing is that we use the same modeling technique to describe functions and components. Both are described by a syntactic interface and an interface behavior that is modeled, for example, by a state machine. Functions are part of a function architecture and components are part of a component architecture. Both structures are constructed by parallel composition of function specifications or component specifications respectively. Both structures address the same scope (the current *system under development*). However, methodically there is a huge difference between a function and a component. A function describes distinct behavior from a user's point of view. That means a function behavior is defined, in general, over interactions between the system and its environment. A component, on the contrary, is a part of a solution structure that realizes/implements the functions. That means a component may have a system-internal interface that does not correspond to interactions between the system and its environment at all. For example, we consider the *airbag function* as a function of a car control system because it describes a set of interactions between the control system of the car and its environment. If, for the realization of the airbag function, it is necessary to aggregate a set of computed values at some point, we may model this aggregation as a solution component. This component however, has no direct interactions with the environment. It is purely system-internal and describes a part of the solution concept. A user would never consider this value aggregation component as a function of the system. An extreme case of this differentiation can be observed when considering a multifunctional system that is implemented by a *layered architecture* (cf. [Taylor et al., 2009]). In such a system, each function crosscuts the whole component architecture. Functions and components are in an *n:m* relation (see Figure 5.17).

Relating Function Specifications and Component Architecture

As both, the components in the component architecture and the function specifications in the function architecture rely on the same modeling techniques it is possible

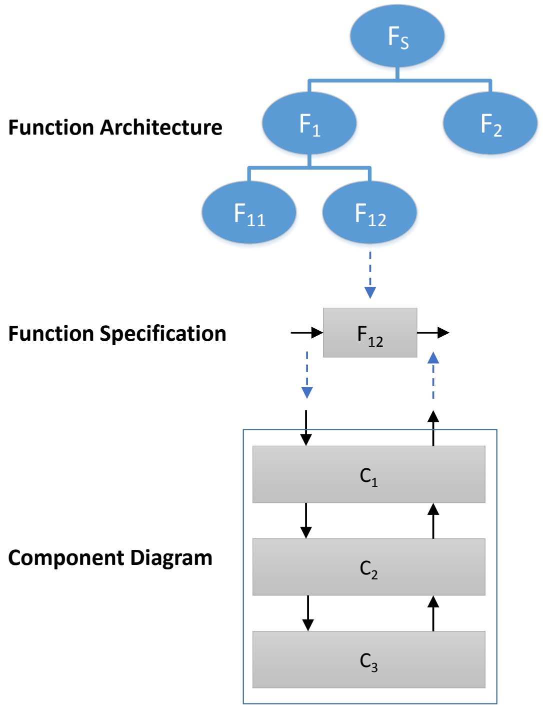


Figure 5.17: In a *layered architecture*, functions crosscut the whole architecture. Functions and components are in an *n:m* relation.

to relate them to each other by their interface specification. However, since the syntactic interface of a function specification and the component architecture may not match exactly, for example due to different levels of abstraction, they possibly need to be translated. For this purpose, we create a *refinement specification* that defines the relation between the interface behavior of the component architecture and that of a function specification.

Refinement Specification A refinement specification translates between interactions of a function specification and interactions of a component architecture. Therefore, the input/output channels of the component architecture are related to input/ output channels of a function specification. Typically, beyond simple channel mappings, a conversion of data types has to be carried out to compensate for different levels of abstraction (cf. [Mou and Ratiu, 2012]).

To relate a function specification to the component architecture, we map the inputs of the function specification to the inputs of the component architecture (representation function) and the outputs of the component architecture to the outputs of the function specification (interpretation function). Figure 5.18 illustrates this mapping. In model-based testing these refinement functions are also known as abstraction and concretization functions [Prenninger and Pretschner, 2005].

Relating Modes and Architecture A special case of refinement is the refinement of mode inputs or mode outputs of a function specification. These ports of a function specification do not have a direct counterpart at the interface of the component

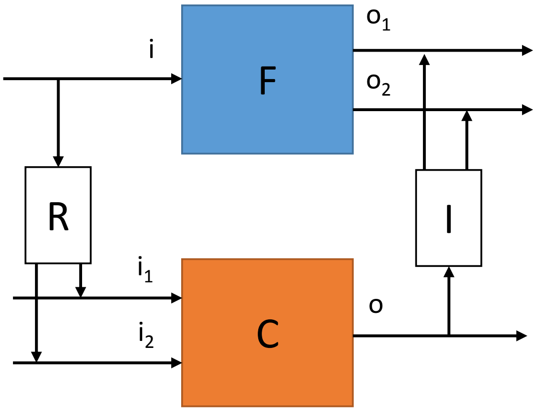


Figure 5.18: A *refinement specification* translates inputs of a function specification F to inputs of a component C by a representation function R, and outputs of the component to outputs of the function specification by an interpretation function I.

architecture because they model the interaction between functions and are thus not part of the black-box interface of the composed system specification. To be able to map the mode channels of a function specification to the component architecture, we need the concept of a *functional white-box diagram*.

Functional White-Box Diagram A functional white-box diagram is a projection of the component diagram to a subset of components that contribute to the realization of one function. For the creation of a functional white-box diagram, this relation between functions and contributing components needs to be defined or computed beforehand. Figure 5.19 gives an example of a functional white-box diagram. In the figure, the channels are arranged in a way that channels of the original black-box interface of the system are denoted on the left and right border of the functional white-box diagram. We call these *primary channels*. Channels that are internal channels in the original component diagram are denoted on the top and at the bottom of the functional white-box diagram. This notation helps us to explain the mapping between a function and its functional white-box diagram in the following.

A functional white-box diagram itself is again a component diagram specifying an interface and an associated interface behavior. Thus, we can define a refinement relation between a function specification and its functional white-box diagram by mapping primary inputs and outputs of the function specification to primary inputs and outputs of the functional white-box diagram and mode inputs and outputs of the function specification to internal inputs and outputs of the functional white-box diagram. Figure 5.20 shows the refinement of modes for the example of Figure 5.19.

It is important to note that the functional white-box diagram of a function specification is not uniquely determined by the function specification. It is rather a design decision to determine the logical components that contribute to the realization of a function.

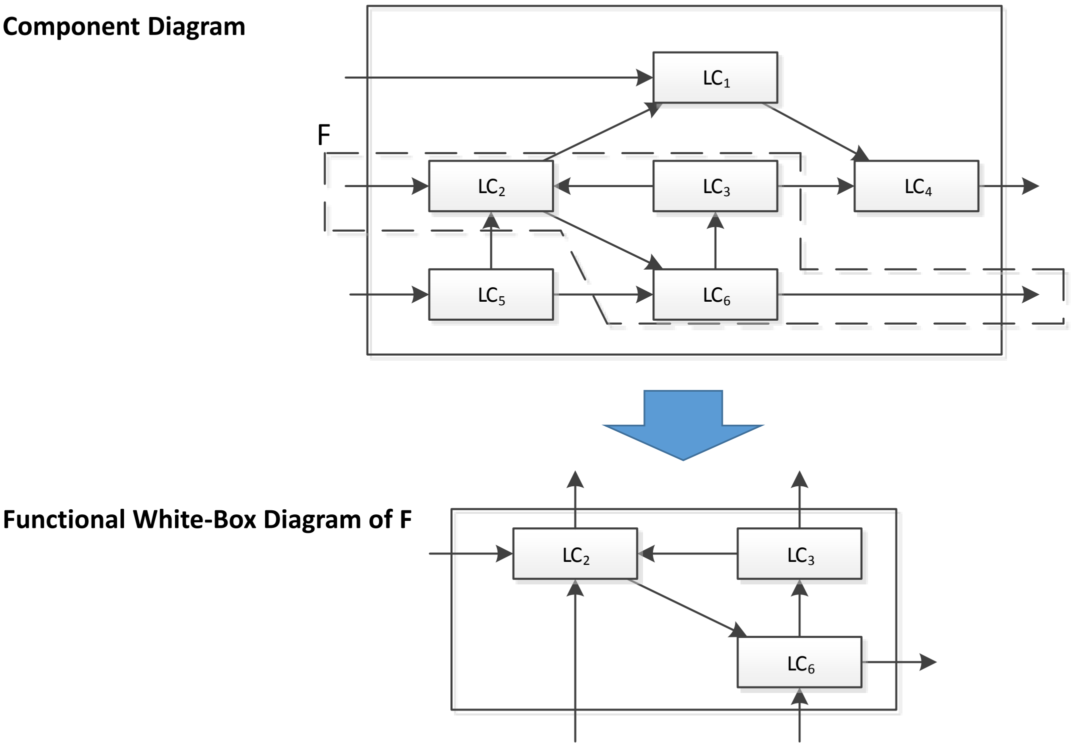


Figure 5.19: Example of a functional white-box diagram as a projection on the component diagram to the function F.

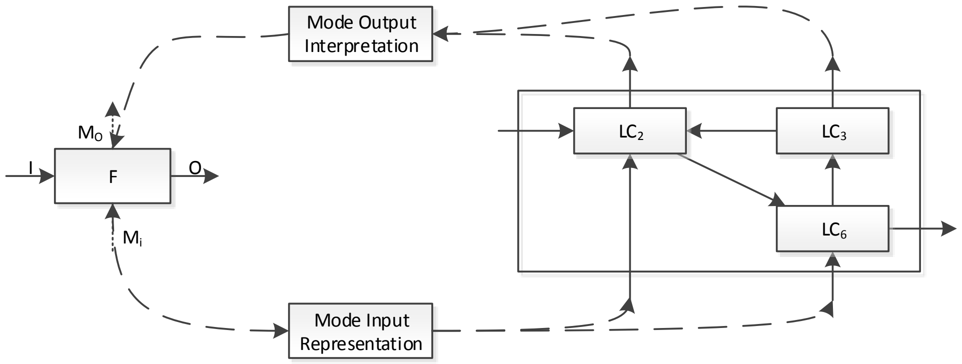


Figure 5.20: Mode refinement relation of function specification F with mode channels M_I and M_O.

5.1.5 Summary: Artifact Model

In this section, we introduced an artifact model that relates the concepts of our semantic model to model types that we use to describe them. The relations between the semantic concepts are reflected and expressed by the model types. We defined these model types to systematically develop, capture, and analyze the semantic content of the system modeling theory. This artifact model integrates a formal framework for the specification of multifunctional systems into model types that are used in our model-based requirements engineering methodology. The artifact model is independent of the development process by which the models are created. The following section of this chapter details the role of these artifacts in a development process.

5.2 Process Integration

In this section, we focus on the role of the presented artifacts in a development process and the benefits for different development activities. We illustrate the role of the artifacts and the activities associated with them by relating them to the V-Model development lifecycle. We consider the function architecture with associated function specifications together with the mode model as system specification and call this the *functional architecture*. Figure 5.21 shows a V-Model including this functional architecture that serves as system specification, and development activities related to it. The presented artifact model supports these activities, for example, by enabling (semi)automated analysis of them. We show how the presented artifacts support the validation of the system specification (Section 5.2.1), the assurance of consistency within the system specification (Section 5.2.2), the verification of the system architecture against the system specification (Section 5.2.3), the verification of the integrated system against the system specification (Section 5.2.4), and the possibility of integrating system specification and safety analysis (Section 5.2.5).

5.2.1 System Specification Validation

Validating a system specification means assuring that the specified requirements correspond to the actual needs of the stakeholders. The presented artifact model supports this activity by enabling a validation of the system specification by simulation.

The purpose of simulating the system specification is to ensure that the specified requirements are complete and correct with regard to user intentions [Davis, 1982]. Specification simulation requires an executable model of requirements (i.e., a prototypical representation) and embeds this executable model into a simulation environment. In the literature, there are two notions of simulation: *Simulation* and *animation*. Simulation is an execution of the software model, whereas animation is a visualization, in a suggestive form, of the simulated model in its environment [van Lamsweerde, 2009]. Davis [1982] compares a simulation with an interactive terminal (or console) that presents some textual output and accepts inputs by the user. However, in some cases, this type of simulation is also referred to as requirements animation [Kazmierczak et al., 1998; West and Eaglestone, 1992]. On the other hand, Pohl [2010] presents what the author refers to as requirements *animation* or *prototyp-*

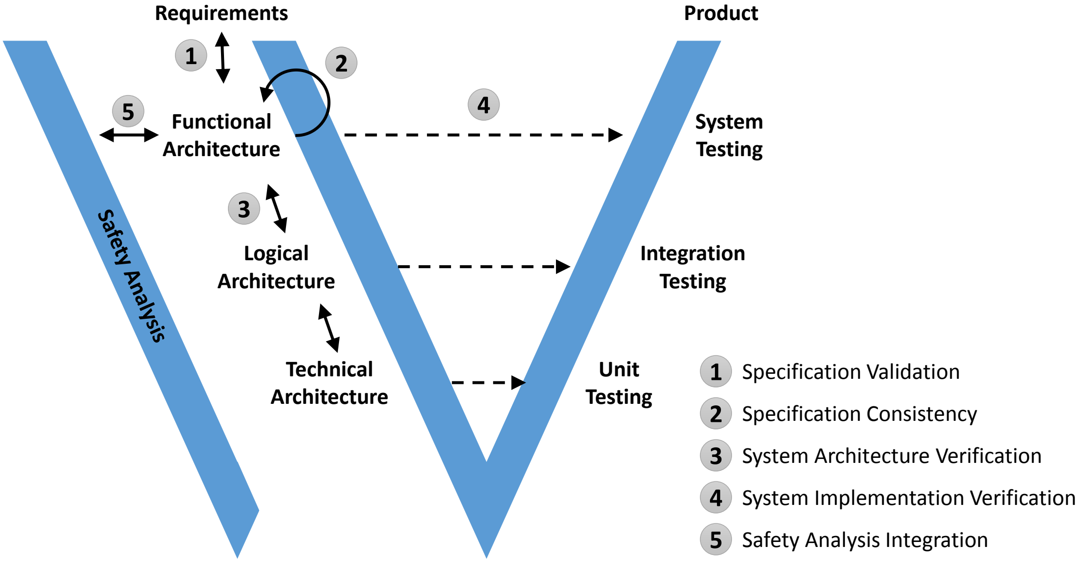

Requirements
Product
1
2
5
4
Functional Architecture
System Testing
3
Safety Analysis
Logical Architecture
Integration Testing
Technical Architecture
Unit Testing
1 Specification Validation
2 Specification Consistency
3 System Architecture Verification
4 System Implementation Verification
5 Safety Analysis Integration

Figure 5.21: The role of the functional architecture serving as system specification in the development process.

ing. This approach is akin to the graphical simulation model presented by Acosta et al. [1994]. Hence, the animation approaches taken by Pohl [2010] and Acosta et al. [1994] mainly describe an approach to visualize the system behavior.

According to Jones, requirements validation by specification simulation is the most effective method to reveal errors in specifications [Jones, 1998]. For specification simulation, an executable representation of the system specification and its environment is required. We therefore need a formal specification of what we want to check for adequacy. This specification must be executable or transformable into an equivalent executable form (cf. [van Lamsweerde, 2009]). In our methodology, this means that the function specifications have to be defined in terms of an executable description technique such as state machines, code specifications, or tabular expressions.

The application of our RE artifact model allows performing specification simulations under the stated preconditions. Scenario descriptions and MSCs are used to specify validation scenarios, which can be discussed with the stakeholders. Scenario enactment tools can execute these validation scenarios and show the results of interactions along timelines in a visual form.

The function specifications of the function hierarchy can be simulated if they are described by executable specification techniques. Due to the specified composition operator for function specifications, it is possible to simulate only parts of the function hierarchy, showing also the results of the interplay between different functions.

5.2.2 System Specification Consistency

One advantage of the introduced artifact model is that it explicitly defines the relationship between model types and description techniques. In some cases, the consistency of these relationships can be assured by precise analysis techniques. Especially, if these analyses can be performed (semi)automated, we expect a great impact on artifact consistency over the whole development lifecycle. In the following, we introduce a set of activities to ensure a consistent system specification.

Manual Review of Use Case Tables, Scenario Descriptions, and MSCs As described in the artifact model, use case tables and scenario descriptions are structured but still informal artifacts. That means the ability to check consistency between use case tables, scenarios descriptions, and their transformation to MSCs automatically is limited. Therefore, manual reviews are necessary to check the consistency of these three artifacts. The properties that need to be assessed in the reviews directly result from the artifact relations defined in the artifact model, for example:

Between a use case table and a scenario description:

- Does every scenario description start with the same precondition and trigger as stated in the use case table?
- Does every scenario description end with the minimal guarantee as stated in the use case table?
- Are the actors of a scenario description step mentioned in the use case table?

Between a scenario description and its MSC:

- Is every step of the scenario description translated to at least one message in the MSC?
- Are all actors of the scenario description translated to roles in the MSC?
- Is the source of an MSC message the actor of the associated scenario step?

Note that each change in a use case table, a scenario description, or an MSC has to undergo a (regression) review. The process of deriving formalized requirements models from use cases tables may also be assisted by automated approaches. Sawant et al. [2014], for example, introduce an approach to derive process diagrams and an ontology from textual use case descriptions using methods based on computational linguistics. The derived process diagrams are very close to scenario descriptions and thus may be translated into MSCs. However, these approaches rely on a consistent use of natural language and may even restrict its use. Therefore, we advise to perform manual reviews of this step even if (semi)automated approaches are used for deriving formalized requirements models.

Manual Review of Textual Property Descriptions and Interface Assertions Similar to the validation of use cases and scenarios against their formalization as MSCs, informal textual property descriptions have to be validated against their formalization as interface assertions. This process can also only be automated to a limited degree, which is why manual reviews are necessary to check the validity of the formalization. The properties that need to be checked in the reviews directly result from the relations defined in the artifact model, for example:

- Is every predicate (i.e., atomic assertion) of the formalization also mentioned in the informal textual property description?

- Is every assertion that is mentioned in the textual property description translated to a predicate (i.e., atomic assertion) of the formalization?
- Is every execution sequence implied by the temporal logic formula intended by the informal textual property description?
- Is every intended execution sequence of the informal textual property description implied by the temporal logic formula?

Especially the last two properties are hard to check without a structured procedure that may be supported by an appropriate tooling. Kof and Penzenstadler [2011], for example, propose a tool-supported validation of formalizations based on feedback. A tool generates exemplary execution sequences implied by the formalization and an analyst assesses whether these are intended or not.

Automated MSC Feasibility Checks between MSCs and Function Specifications As described in the artifact model, a function specification must *fulfill* all MSCs related to that function specification. More precisely, an MSC provides a sequence of messages that are exchanged between the system and its environment. The function specification must reflect this sequence of messages, i.e., it must be a valid execution trace of the function specification. We automatically check this property by an *MSC Feasibility Check*. This check transforms the sequence of messages specified by the MSC into a logical proposition and uses a model checker to evaluate whether the function specification is a model for the logical proposition. If so, the model checker provides an execution of the function specification as evidence. The transformation of MSCs to logical propositions is described, for example, by Broy [2005].

Automated Model Checking between Interface Assertions and Function Specifications To check whether a function *fulfills* all interface assertions related to it, we apply model checking techniques (cf. [Clarke et al., 2005]). This is possible since temporal logic formulas are used to describe the behavior of an interface assertion, and function specifications can be transformed into a labeled transition system [Keller, 1976]. Similar to the MSC feasibility check, the model checker evaluates whether the function specification is a model for the temporal logic formula. If this is the case, the function specification is consistent with the interface assertion. If not, either the function specification or the interface assertion is incorrect and must be changed.

5.2.3 System Architecture Verification

Given a refinement relation specification between a function specification and the component architecture, we can automatically verify that the behavior of the component architecture is a refinement of the function specification.

Automated Refinement Test between Function Specification and Component Architecture The function specifications serve as partial specifications for the system behavior. In Section 5.1.4, we showed how this specification is connected to the component architecture of the system by a refinement specification. A *refinement test* ensures that the system behavior actually refines the specification (cf. [Mou and

Ratiu, 2012]). In a refinement test, test cases are automatically generated from the function specification. Each test case provides a valid input/output relation given by the function specification. The test cases are then automatically translated to test cases for the component architecture by means of the refinement specification as described in Section 5.1.4. The test cases are executed on the component architecture and the output results are afterwards translated back to the function specification level and compared to the output results of the original test case. If the results of the test case execution deviate from the function specification, the component architecture is not correct with respect to the function specification.

5.2.4 System Implementation Verification

The test cases derived from the function specifications are not only used to test whether the component architecture is a correct refinement of the system specification, but they are also used to test the actual implemented system. This resembles the idea of model-based testing, which is to use explicit behavior models to describe the intended behavior. Traces of these models are interpreted as test cases for the implementation: input and expected output. The input part is fed into an implementation (the system under test), and the implementation's output is compared to that of the model, as reflected in the output of the test case [Prenninger and Pretschner, 2005].

Our artifact model supports model-based testing in several ways. First of all, we use test cases derived from function specifications to test the correct implementation of single functions in an integrated system. The test case generation and execution can be performed completely automated based on the executable function specifications and a refinement relation to the implemented system.

In addition to those function tests, also function integration and dependency tests can be performed. These tests ensure that the integration of a set of functions in one system yields the desired properties and behavior. Due to the explicit modeling of function dependencies by means of mode channels, the test case inputs trigger also possible mode switches that may influence the other functions, which then react according to the switched mode. The resulting outputs reflect specified behavior that results from the integration of a set of functions. The specified function dependencies can also be exploited to create test cases, which employ a specific set of mode switches or affect a large set of different functions.

In our methodology, test cases are derived directly from the specification of requirements and thus can be traced back to specific properties or use cases. A change in the requirements accounts for a change in the specifications. In a process, where test cases are derived from the specification models, the test cases can be updated automatically. Thus, specification and tests are always consistent. A third point is that test coverage metrics can be applied to the coverage of specification models. Thus, it is possible to assess the amount of requirements that are assured by a test case.

5.2.5 Safety Analysis Integration

Functional safety assurance is the process of proving that the system under development can do no harm to its environment. Many embedded systems are safety-critical as they directly influence physical objects that may exhibit hazardous behavior to their environment. There are several applicable standards for developing safety-critical electronic systems, such as the IEC 61508-3 [IEC, 1998] or the ISO 26262 for the automotive sector [ISO, 2011]. They prescribe activities and techniques that have to be used to build a safety-critical system. By this, the process aims at a clear argumentation, why a system cannot harm its environment. A safety case, for example, is a structured line of arguments, which shows that the system under consideration is safe. It addresses the difficulty of combining a large variety of information needed to form this argument [Wagner et al., 2010].

A difficult task during the functional safety assurance procedure is to assess the consequences of a failure (both sporadic hardware failures and systematic software failures) to the behavior of the software and the resulting behavior of the system. Therefore, it is necessary to have a clear understanding of system behavior in the presence of failures.

The model types and artifacts presented in this thesis can be extended by modeling failures explicitly as part of the behavior. A safety requirements, such as "event a and event b must never occur simultaneously", is formalized as an interface assertion and needs to be fulfilled by a function specification. Thus, function specifications not only capture the desired functionality but also describe conditions and constraints that have to be fulfilled to prove that the overall system is safe. Furthermore, it is possible to enrich function specifications to also model and specify the behavior under the influence of failures as illustrated in the following example: A *safety goal* describes the desired behavior of a system in the presence of a failure. This safety goal is specified by a set of observations between the system and its environment. To specify this safety goal, it is necessary to extend the interface description of the system by additional failure stimuli. We get an interface description that expresses relations between operational inputs and outputs as well as information about failures and their consequences for the behavior.

The introduction of a mode model supports the specification and elicitation of hazardous situations, especially when these arise from the interplay of different functions of a system. Based on a mode model, safety goals, such as the mutual exclusion of two modes, can be formulated and it can be checked whether these conditions are fulfilled by the composition of all functions in a system.

Enriching a purely functional specification model with information about failures leads to the notion of *rich specifications* [Damm et al., 2005]. In rich specifications, extra-functional aspects, such as failures, timing conditions, or availability information (cf. [Junker and Neubeck, 2012]), are added to the function specification. This makes analyses, such as functional safety assessments, more precise and expressive.

5.2.6 Summary: Process Integration

In this section, we integrated the artifact model that was introduced in Section 5.1 into the V-Model development lifecycle. This integration illustrates the role of the artifacts in development activities that profit from the application of the artifact model. More specifically, we showed how the artifacts enable the assurance of a valid and consistent system specification, the verification of the system architecture against this specification, and the support for safety analyses. By this contribution, we ensure that our artifact model has clearly defined relations to subsequent activities in a development process. In the problem statement of this thesis (see Section 1.2), we stated this integration as an important property for the adoption of the artifact model in industrial practices. The artifact model and its process integration together form our RE methodology.

In a concrete development process, the RE methodology needs to be instantiated based on the system to be developed and its development context. This instantiation has an impact on the artifact construction and analysis procedures that are necessary and possible. In the following, we present two case studies that show and evaluate two instantiations of our RE methodology (1) in a scenario-driven requirements engineering context and (2) in a property-driven requirements engineering context. These case studies explore the possibility to apply our artifact model for different styles of eliciting and documenting requirements. We will see that, depending on the style, the artifact construction and analysis procedures differ, but for both styles, we were able to end up with a formal and model-based functional system specification.

5.3 Case Study: Scenario-driven Requirements Engineering

In a concrete development process, our proposed RE methodology needs to be instantiated to fit the characteristics of the system under development and the demands of the development context. In a first case study, we instantiate our methodology in a scenario-driven RE context and assess its applicability, benefits, and limitations. In a scenario-driven RE approach, requirements are mainly specified by eliciting desired usage scenarios. We decided to take a scenario-driven RE approach for the study object based on the available information about the system.

5.3.1 Study Context: Scenario-driven Requirements Engineering

Scenarios have attracted considerable attention in RE. As discussed by Sutcliffe [2003], scenarios play an important role to stimulate the imagination of the designers. Thus, they help in guiding thoughts, and supporting reasoning in the design process. Scenarios cannot explicitly guide a designer towards a correct model of the desired system. However, they play a key role to start all modeling and design [Misra et al., 2005]. The main purpose of developing scenarios is to stimulate thinking about possible occurrences of these, assumptions relating these occurrences, possible opportunities and risks of a scenario, and courses of action [Jarke et al., 1998].

The advantage of a requirements engineering approach based on scenarios lies in the way the scenarios ground argument and reasoning in specific detail or examples, but the disadvantage is that being specific loses generality. Scenarios in the real world sense are specific examples. The act of analysis and modeling is to look for patterns in real world detail, then extract the essence, thereby creating a model [Sutcliffe, 2003].

For a scenario-driven requirements engineering, we instantiate our model-based RE methodology in a way that, starting from functions described by use case tables, a stepwise formalization of functional requirements via scenario descriptions and MSCs is enabled, which are summarized in function specifications that are finally linked to the component architecture of the system.

5.3.2 Goal

This case study follows the goal of evaluating the applicability, benefits, and limitations of instantiating our model-based RE methodology in a scenario-driven requirements engineering context.

5.3.3 Study Object

We applied the methodology during two consecutive master-level practical courses on model-based engineering for students at our university. The task of the students was to develop the control software for a canal monitoring and control system (CMCS). We took the requirements description of this system from a public case study that was used in the 2011 *Workshop on Model-Driven Requirements Engineering (MoDRE)*[3]. The only available source of requirements was a requirements specification document that consisted of a system overview, two descriptions of desired usage scenarios, and a domain model. We classified this as a scenario-driven RE context.

The Canal Monitoring and Control System

The system's purpose is to control a system of canals on which ships are cruising. To surmount a difference in water level, a canal is equipped with a lock. A lock consists of two lock gates and two valves that are used to balance the water level. A second component in the canal system is a low bridge. In order for a ship to pass the bridge, it needs to be opened.

The CMCS tracks the ships that cruise on the canals and controls bridges and locks such that ships can pass. In the original requirements document, the system was meant to handle a flexible canal topology. However, to simplify the development task, we fixed a canal topology with only one lock and one bridge. Furthermore, we included only two ships into the case study. The topology is depicted in Figure 5.22.

5.3.4 Study Execution

The overall 15 students that participated in the two courses were divided into two groups in both courses. The first group took the role of the requirements engineers, while the second group was responsible for the system architecture (the architects). The task of the requirements engineers was to document functional requirements and to formalize them as use case tables with scenarios descriptions, MSCs, and function specifications. They additionally created the system specification in terms of a function architecture and a mode model. The architects developed the component architecture and created the refinement specifications between the functions and the com-

[3] http://cserg0.site.uottawa.ca/modre2011/

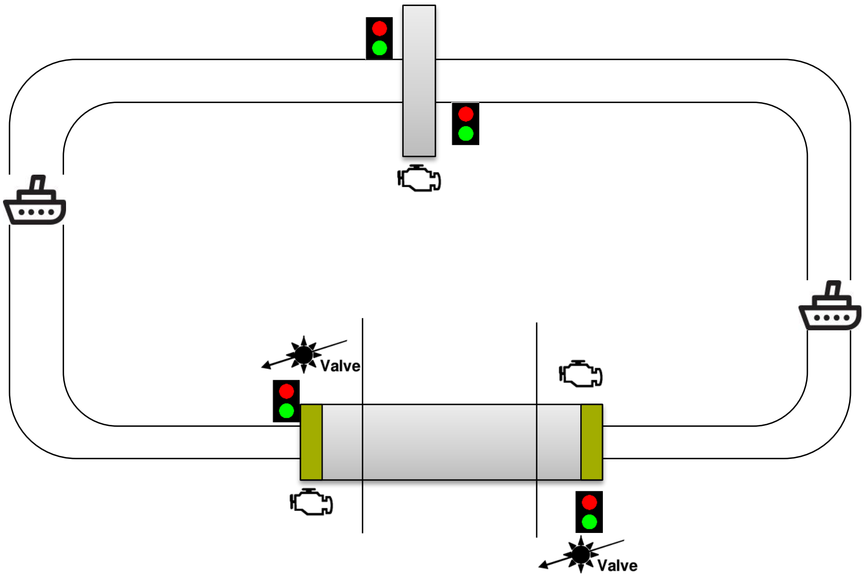


Figure 5.22: The topology of the canal system includes a lock with two lock gates (bottom of the figure) and a low bridge (top of the figure). The low bridge and the lock are equipped with traffic lights, motors, and valves.

ponent architecture as described in Section 5.2.3. Both groups together performed the refinement tests and, in case of test failures, decided on the actions to be taken. The created artifacts were continuously reviewed by the course supervisors.

The system was developed incrementally. As two main functions, we identified controlling the bridge and controlling the lock. The requirements engineers and the architects worked in parallel. While the requirements engineers documented and formalized the requirements, the architects came up with a first rough architecture.

5.3.5 Instantiation of the RE Methodology

To apply the artifact model in a scenario-driven requirements engineering context, we instantiated and tailored the artifact model, and provided a concrete representation. Based on the requirements document given as input for the case study, we used only a subset of model types from the artifact model described in Section 5.1. The process we took in this case study to create and verify artifacts is depicted in Figure 5.23. The figure shows the order in which we created artifacts of the artifact model (solid arrows) and checked their consistency (dashed arrows). A detailed description of this study execution process is given in the following.

Step 1: To set the scope of the project, the course supervisors defined a *function list* that contained the functions *Pass Bridge* and *Pass Lock*. We extracted those as functions because they were stated as “use cases” of the system in the original requirements document.

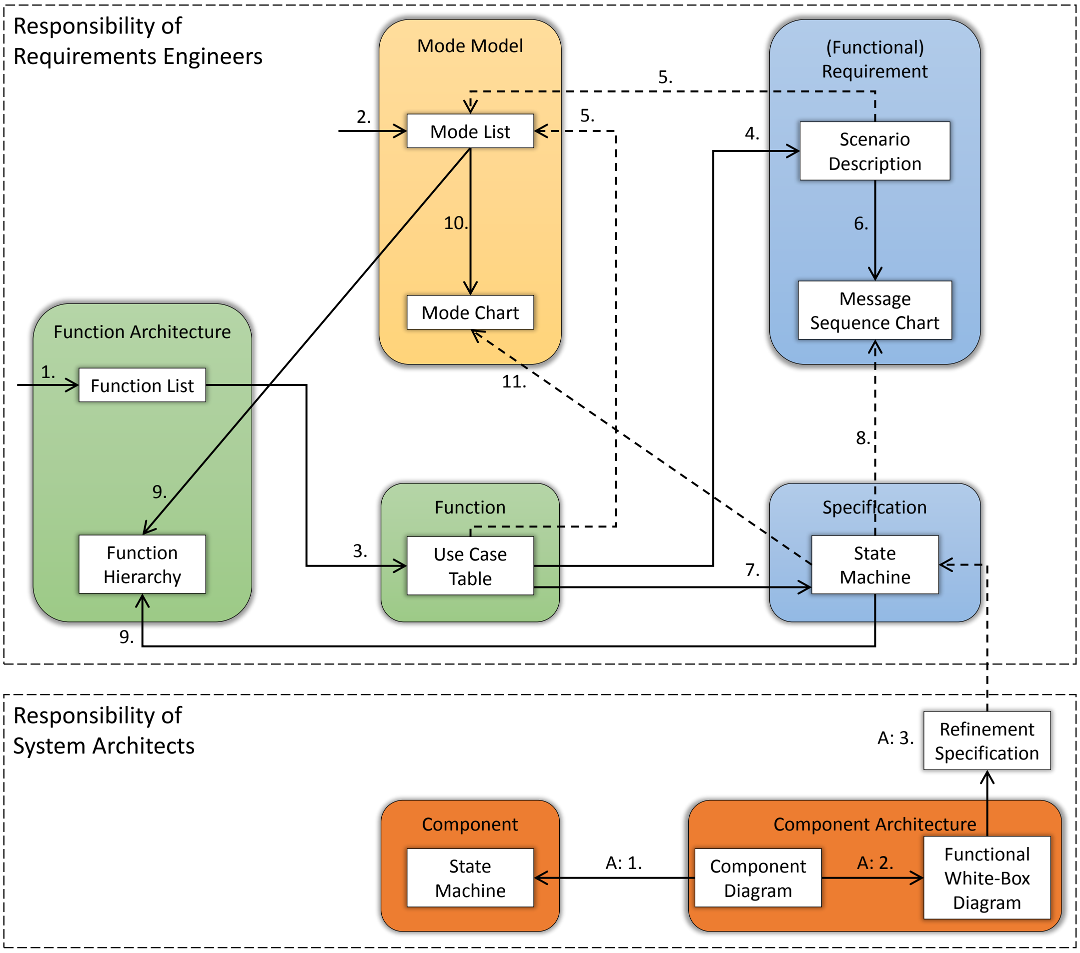
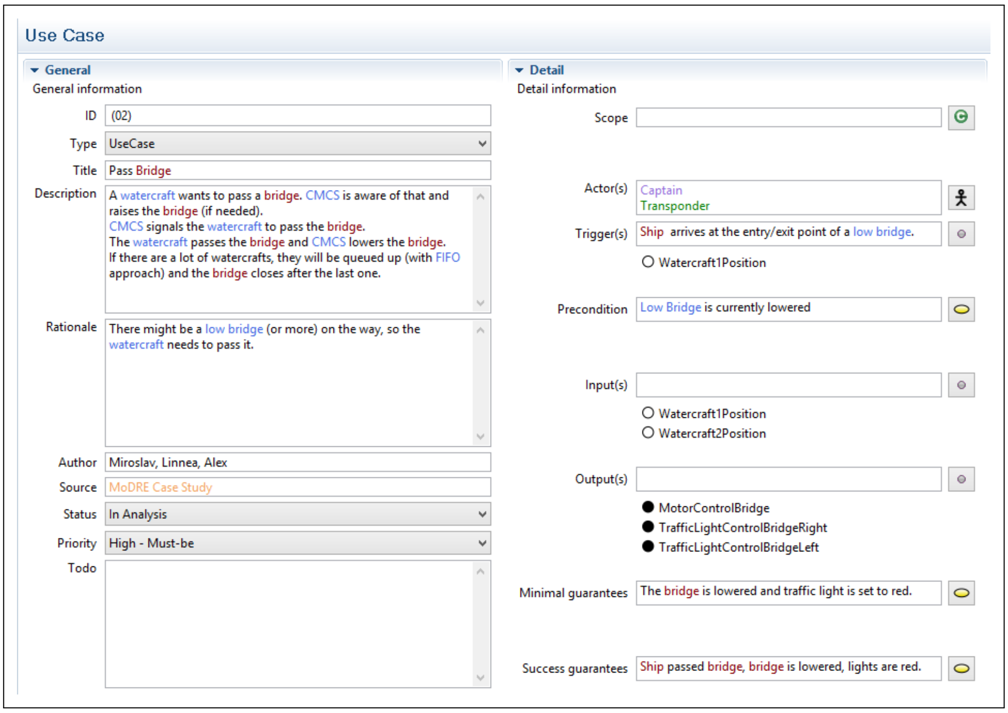


Figure 5.23: Study execution process taken in the scenario-driven case study.



Use Case

General
General information

ID: (02)
Type: UseCase
Title: Pass Bridge
Description: A watercraft wants to pass a bridge. CMCS is aware of that and raises the bridge (if needed).
CMCS signals the watercraft to pass the bridge.
The watercraft passes the bridge and CMCS lowers the bridge.
If there are a lot of watercrafts, they will be queued up (with FIFO approach) and the bridge closes after the last one.
Rationale: There might be a low bridge (or more) on the way, so the watercraft needs to pass it.
Author: Miroslav, Linnea, Alex
Source: MoDRE Case Study
Status: In Analysis
Priority: High - Must-be
Todo:

Detail
Detail information

Scope:
Actor(s): Captain
Transponder
Trigger(s): Ship arrives at the entry/exit point of a low bridge.
○ Watercraft1Position
Precondition: Low Bridge is currently lowered
Input(s):
○ Watercraft1Position
○ Watercraft2Position
Output(s):
● MotorControlBridge
● TrafficLightControlBridgeRight
● TrafficLightControlBridgeLeft
Minimal guarantees: The bridge is lowered and traffic light is set to red.
Success guarantees: Ship passed bridge, bridge is lowered, lights are red.

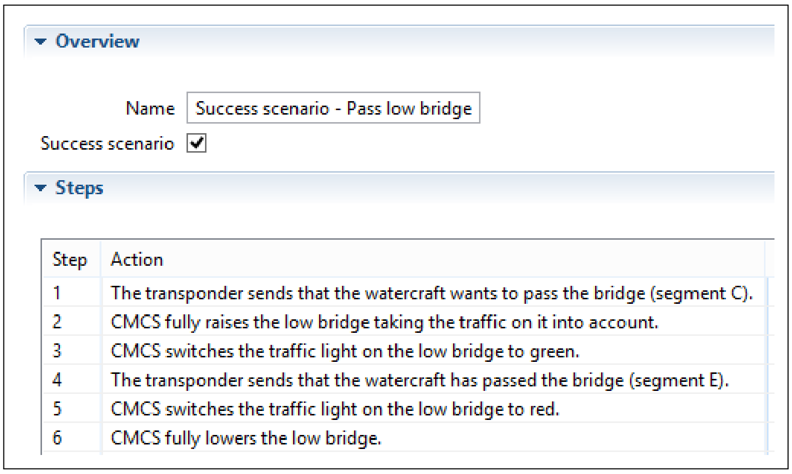
Figure 5.24: *Use case table* for the function *Pass Bridge*.

Step 2: The first step that the students performed was to define a *mode list* from the given input document. For this purpose, the students examined the input document and searched for information that characterize operational states of the system or its environment. They transformed this information to items of the mode list defining modes and their mode values.

Step 3: As next step, the students transferred the information given in the input document with respect to the two functions into *use case tables* and documented them in the tool we used. Figure 5.24 shows the use case table for function *Pass Bridge*.

Step 4: From the use case tables, the students started to define *scenario descriptions* that describe exemplary usage scenarios related to the functions. Usually, this included one success scenario description and a set of alternative scenario descriptions. Figure 5.25 shows an example of a success scenario description for the function *Pass Bridge*.

Step 5: At this point in the process, the students performed a first consistency check between the artifacts they created. They compared the use case tables and scenario descriptions with the mode list they created in step 2 to find references between them. For the use case tables, this includes a relation of the statements in the *trigger*, *precondition*, and *guarantee* fields to the modes of the mode list. In the scenario descriptions, each step was investigated for conditions or actions related to a mode of the mode list. If necessary, the students extended the mode list and aligned the



Name: Success scenario - Pass low bridge

Success scenario ☑

Step	Action
1	The transponder sends that the watercraft wants to pass the bridge (segment C).
2	CMCS fully raises the low bridge taking the traffic on it into account.
3	CMCS switches the traffic light on the low bridge to green.
4	The transponder sends that the watercraft has passed the bridge (segment E).
5	CMCS switches the traffic light on the low bridge to red.
6	CMCS fully lowers the low bridge.

Figure 5.25: Success *scenario description* for the function *Pass Bridge*.

use case tables and scenario descriptions with the mode list. The tool we used was able to store and highlight these references in the textual descriptions.

Step 6: As a first step towards a more precise description of the desired system behavior, the students formalized the informal scenario descriptions by *MSCs*. Figure 5.26 shows the MSC that formalizes the success scenario description of the function *Pass Bridge*. In an MSC, the interactions between the system (CMCS) and its environment are modeled by messages. The referenced modes are modeled by conditions (hexagons) in the MSC.

Step 7: In addition to the scenario modeling and formalization, the students created *function specifications* for each function based on the use case tables. The purpose of the function specifications is to associate the functions with an executable behavior that reflects the intended purpose stated in the use case table. The students specified the behavior of functions by executable specification techniques such as state machines, table specifications, or code specifications. The syntactic interface of the function specification is predetermined by the MSCs associated to the original function. That means the students derived the input and output ports of the function specification by combining the input and output ports associated with the CMCS system in the MSCs. In cases, where it was too difficult or complex to specify the behavior of an entire function by one state machine, the students had the possibility to break the function down to smaller functional units that were then described by state machines. Figure 5.27 shows the function specification of the *Pass Bridge* function.

Step 8: To check whether the executable function specification actually fulfills the behavior of all message sequence charts associated with the function specification, the students applied an MSC feasibility check to verify the consistency between the function specification and the MSC. The compatibility of the syntactic interfaces of MSC and function specification that is necessary to perform the feasibility check was ensured by the construction approach described in the previous step.

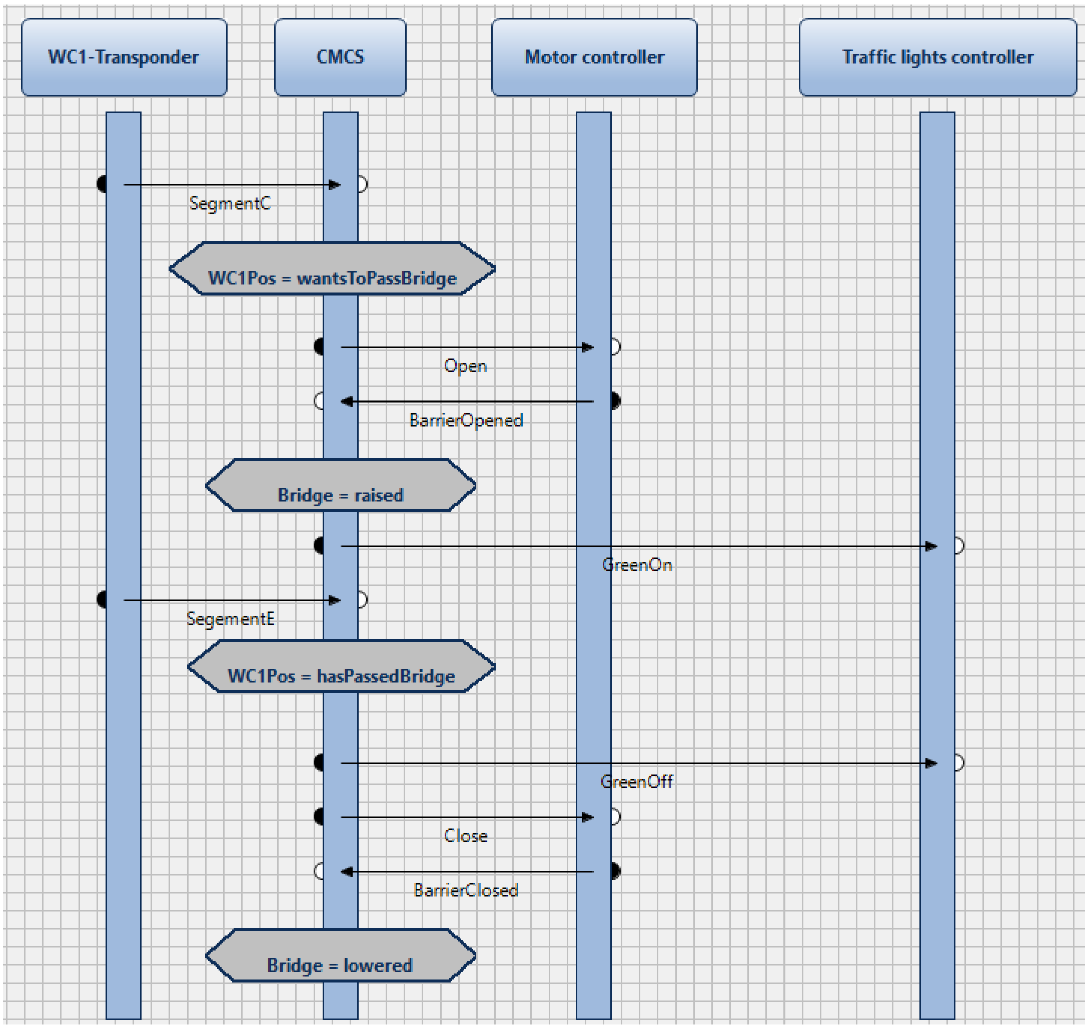


Figure 5.26: Formalization of the success scenario description by a *message sequence chart (MSC)*.

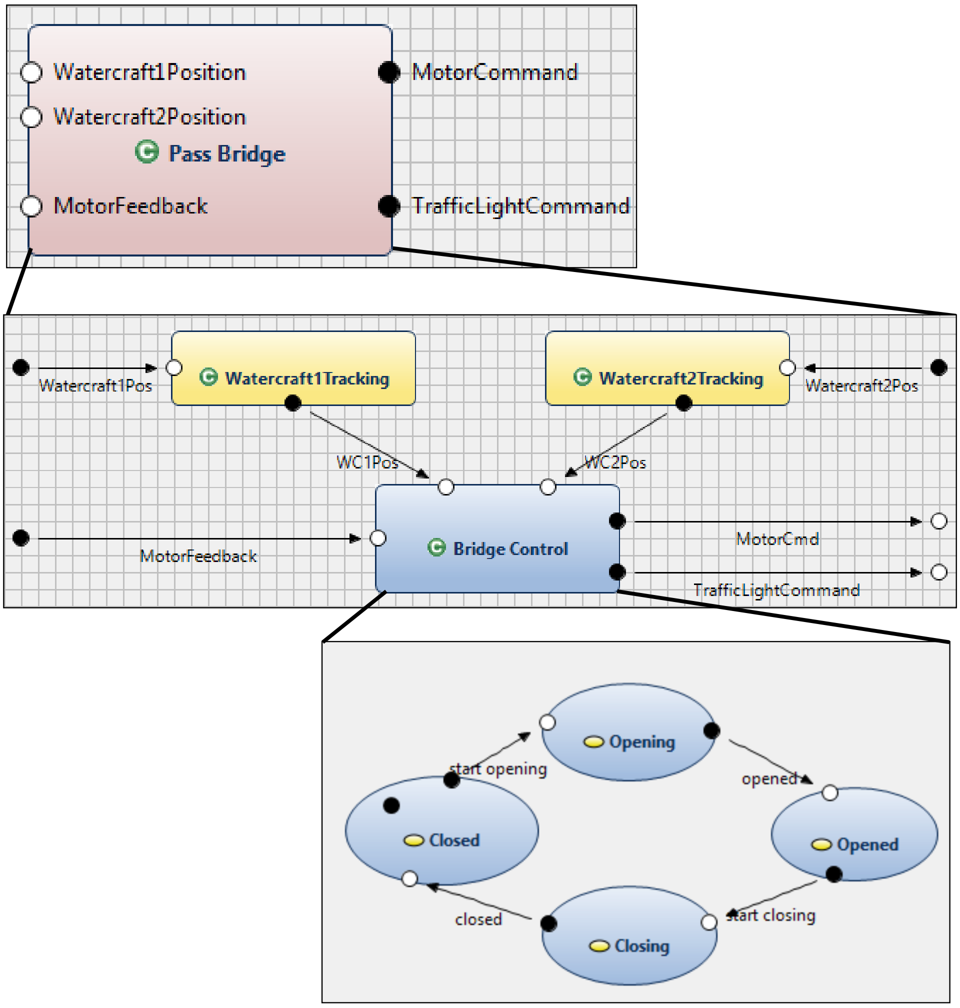


Figure 5.27: Example of a *function specification* with syntactic interface (top) and an associated interface behavior specified by an additional decomposition into subfunctions (mid) and finally state machines (bottom).

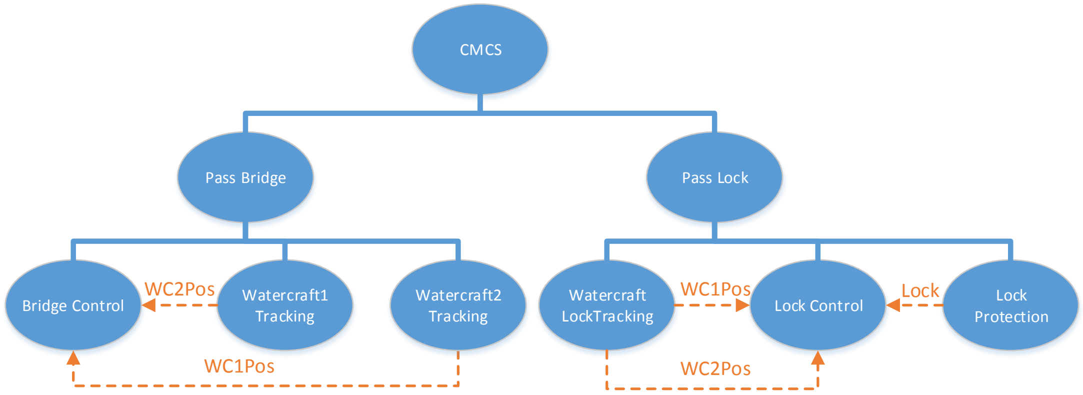


Figure 5.28: *Function hierarchy* of the CMCS system.

Step 9: From the function specifications and the mode list, the students created a *function hierarchy* by associating a function specification with each function of the function list. In case a function was broken down to subfunctions during the creation of the function specification, these subfunctions were added to the function hierarchy. The function dependencies between the functions were related to modes of the mode list. The resulting function hierarchy is shown in Figure 5.28.

Step 10: During the construction of the function hierarchy, the students extended the model list and then started to structure and formalize the modes of the mode list in a *mode chart*. This was performed based on the information given in the original requirements document and the student's decisions. The resulting mode chart is shown in Figure 5.29.

Step 11: The mode chart defines a set of valid sequences of mode configurations that must not be violated by the function specifications. Otherwise, the function specifications are not consistent with the underlying mode model, i.e., either a function specification or the mode model contains an error. To check the consistency, the students compared the possible execution sequences of the mode channels in the function specification with the valid sequences of mode configurations from the mode chart.

Work of the Architects: Concurrent to the specification of requirements and function related artifacts, the architects modeled the component architecture by a *component diagram* (step *A: 1.*) and provided a behavior specification for each component (step *A: 2.*). The architects started working on the basic design (i.e., a high-level component diagram) when the first scenarios were specified. As soon as the requirements engineers had established the first function specifications in step 7, the architects began to define *refinement specifications* between the function specifications and the component architecture (step *A: 3.*). Figure 5.30 provides an extract of the component diagram. A continuous comparison between function specifications and component architecture by means of refinement tests was performed from that point on through all following changes in function specifications or component architecture.

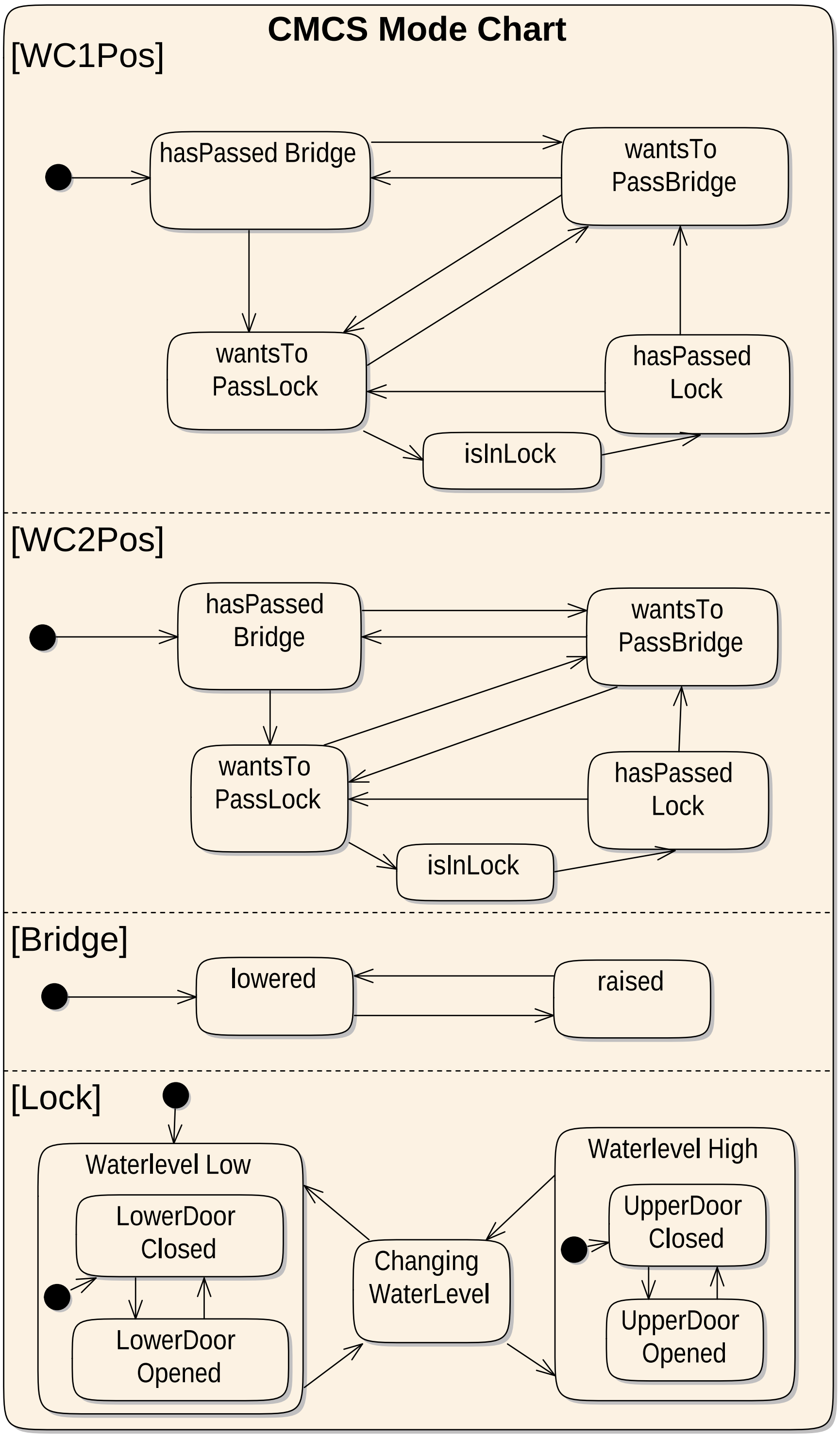


Figure 5.29: *Mode chart* of the CMCS system.

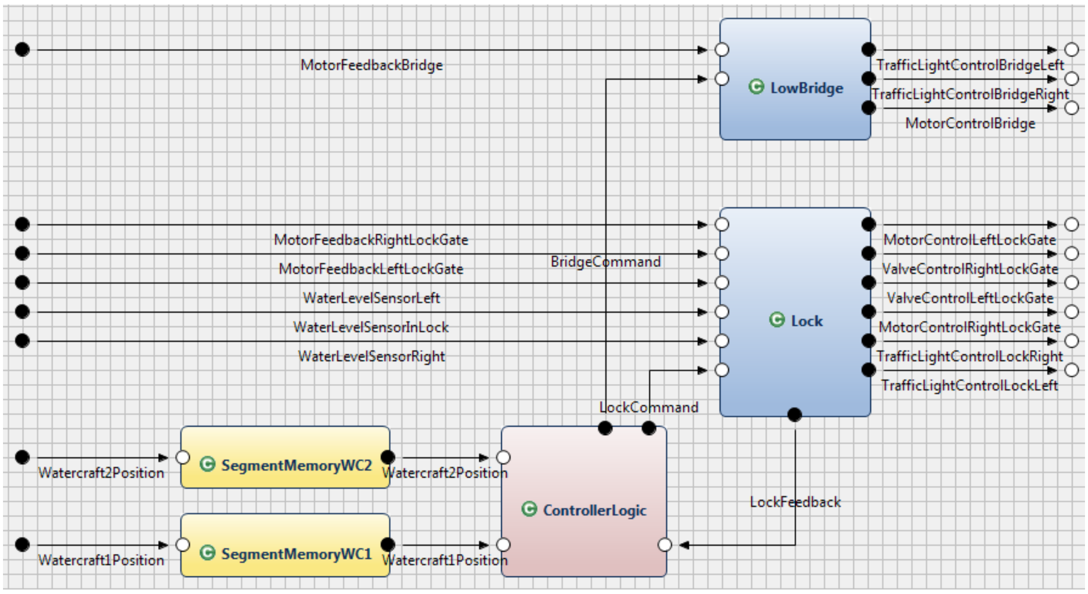


Figure 5.30: Extract of the component architecture represented by a *component diagram*.

5.3.6 Study Results

Reflecting the goal of this study to assess the applicability, benefits, and limitations of integrating the model-based RE methodology in a scenario-driven RE context, we describe our findings structured according to these three subgoals.

Applicability

Overall, the methodology worked well and the system was successfully developed by the students. Due to the high amount of cross checks between the different artifacts of requirements, specification and architecture, we have high confidence in the correctness of the developed system.

Reoccurring Specification Patterns. During the application of the artifact model to the study object, we observed some general patterns that were applied several times to specify reoccurring situations.

One specification pattern was applied in situations, in which a set of function specifications influence a common output in a mutually exclusive way. In our case study, for example, the *Lock Control* subfunction of function *passLock* requires the lock door to be opened when a ship is approaching, whereas the *Lock Protection* function required the lock doors to be closed if the water level in the lock is currently changing. Thus, the corresponding function specifications influence a common output *LockDoorCommand*. This is a problem since the artifact model and the underlying modeling theory require that the set of outputs are disjoint for each function specification.[4] The stu-

[4] The use of the same output in two different function specification may lead to inconsistent specifications since functions always run in parallel and therefore may produce conflicting values for a given output (see [Broy, 2010a]). This is a constraint of the modeling theory compared with other theories that, for example, allow functions to deactivate or interrupt each other (e.g., DFC [Jackson and Zave, 1998], Activity Diagrams [Fowler and Scott, 2000]).

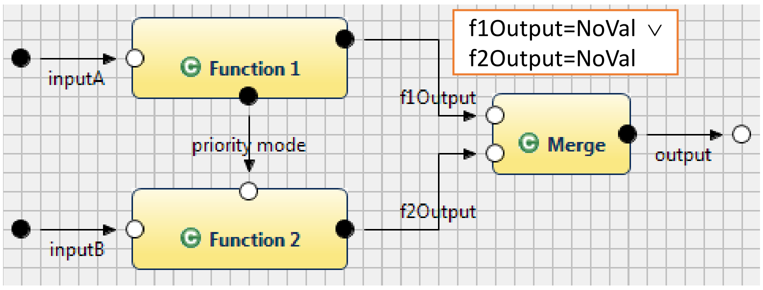


Figure 5.31: A specification pattern describing two function specifications influencing a common *output* in a mutually exclusive way.

dents modeled these situations using a specification pattern shown in Figure 5.31. In this pattern, the set of concurrent function specifications are supplemented by additional *Merge* function specifications (one for each shared output). Instead of directly influencing the shared *output*, the single function specifications forward their specified value for *output* to the *Merge* function specification (via channels *f1Output* and *f2Output*). The *Merge* function specification is equipped with a formal assumption that requires that at most one of the function specifications at a time forward a value, while the others do not specify a value. In the figure, this is expressed by the special value *NoVal* resulting in the formal assumption depicted in the figure. The functions and their interactions have to be specified in a way that this assumption holds globally. In general, this requires the introduction of mode channels (e.g., *priorityMode* in Figure 5.31). In our case study, the students integrated the mode channel *Lock* between *Lock Control* and *Lock Protection* to express a higher priority of the *Lock Protection* function (see Figure 5.28).

The application of this pattern can easily be extended to more than two concurrent function specifications and multiple shared outputs. The advantage of this pattern is that all requirements of a function, including priority over other functions, are captured in that function specification. The *Merge* functions, for which it is arguable whether these actually resemble the original idea of a function as introduced in Section 5.1.3, are reduced to technical entities that carry no functionality that is not already contained in the function specifications themselves. The assumption of the *Merge* function specification enforces that the concurrent function specifications cannot specify inconsistent behavior with respect to the shared output.

Benefits

Correctness of Requirements. The formalization and modeling of usage scenarios as MSCs provided precision and demanded decisions in an early stage of the development process. Therefore, the requirement engineers uncovered several misconceptions regarding the behavior of the environment very early. For example, among the requirement engineers there was a different understanding what kind of information the watercraft sends to the system (current GPS position or current canal segment), which was then decided during the creation of the MSCs.

Concurrent Development of Requirements and Architecture. Due to the precise definition of the artifact relations and their continuous assessment, concurrent development of requirements and architecture was supported by our RE methodology. Changes and refinements within the requirements or the architecture were immediately checked for consistency with the related artifacts. Requirements and architectural elements were linked right from the beginning. The first correctness checks between specification and architecture were applied after one fifth of the project time. In the course of the development, the requirements and the architecture were further refined and formalized allowing more expressive analyses to ensure the correct relation between requirements and architecture. All artifacts were continuously updated and consistent with each other. A typical situation, where our methodology proved beneficial compared with simple informal tracing links, occurred, when the architects had to change the component architecture due to a change in the requirements of one function. In many cases, the changed component architecture fulfilled the altered requirements, however, also introduced bugs with respect to some other function. These bugs were revealed by the refinements tests and led to the revision of other requirements in many cases, showing that information flows not only from requirements to architecture but also in the other direction.

Explicit Design Decisions. Another aspect of the artifact relations is that they explicitly express the design decisions taken during development. Each artifact carries information that originates from information of another artifact. The relation links between the models document the design decisions made at the transition between artifacts and make them explicit. For example, the transition from the function specification to the component architecture is captured by the refinement specification. Rather abstract signals like "a ship approaches a bridge" are translated to concrete logical signals like "BridgeSensor sends a signal". The design decision, to express the event of the requirement by this specific sensor input, is explicitly documented in the refinement specification between the function specification and the component architecture. This was especially helpful for assessing whether a specific requirement is fulfilled by the chosen architecture. The same applies for the transition of an informal step in a scenario description to a formalized sequence of messages in an MSC.

Limitations

Scalability. Our approach aims at a modular specification that allows breaking down the functionality of a system into smaller parts and assessing them individually. It is still an open question whether analysis procedures and checks can be performed in a reasonable amount of time. The previously described system consisted of two functions with an overall of 10 scenario descriptions that were refined into MSCs. Our students were able to model all requirements related to the functions in the original specification by means of our methodology. We additionally elicited 17 further requirements that were first formalized separately by means of temporal logic expressions and then mapped to the two functions. 3 out of the 17 additional requirements could not be formalized as an interface assertion because they were too abstract (i.e., an additional clarification step with the stakeholder is necessary). The most complex function specification of the system had an overall state space of 10,976 states. However, the ability to decompose this function specification to a set of subfunctions made it possible to model this function specification by a set of subfunctions with state machines having a maximum of seven states. The most time consuming automated analyses were the MSC Feasibility Checks that lasted up to 100 seconds for an MSC with 36 messages.

Missed Reuse Opportunities. A problem was that the function specifications got very complex. Therefore, the requirements engineers started to structure them hierarchically, which resulted in the above-mentioned subfunction hierarchies. This seemed to be problematic at first, as apparently the work for structuring the functionality was done twice, in the function specification and in the component architecture. However, we noticed the rationales behind the structuring were quite different between requirement engineers and architects. Where the requirement engineers mainly strove for ease of understanding, the architects had goals such as high reuse or an efficient deployment in mind. Whether parts of the function architecture can be (re)used for the component architecture is a question that needs further investigation.

5.3.7 Threats to Validity

A threat to the *internal validity* is that the notion of correctness in the end was determined by the course supervisors. We had no opportunity to assess the validity of the developed system with the actual stakeholders. We tried to mitigate this threat by focusing on the given requirements document and trying to be as close to the given requirements document as possible. Another threat to the internal validity is that the used requirements document was designed to be used as a basis for discussion in an international workshop. It therefore might not describe a real system and the document might be created with the goal to facilitate discussion rather than resembling real requirements. We have not discussed this threat with the authors of the requirements document; however, we have not found any evidence or suspicious requirements that support this threat.

5.4 Case Study: Property-driven Requirements Engineering

To broaden our experience in applying the proposed RE methodology, we instantiated the methodology in a second case study in a property-driven RE context and assessed its applicability, benefits, and limitations. In a property-driven RE context, requirements are mainly specified by independent statements about desired system behavior. Sometimes, these statements are structured with respect to some criteria. We classified this second case study as having a property-driven RE context due to the characteristics of the available information about the study object.

5.4.1 Study Context: Property-driven Requirements Engineering

Specifying requirements by independent properties has positive impacts on the RE process [van Lamsweerde, 2001]:

- Attaining completeness of the requirements. Properties can support the right criterion for the sufficiency of the completeness.

- A property refinement tree can provide links to trace technical requirements from strategic concerns. This helps to explain the requirements to stakeholders.
- At the time of requirement elaboration, requirements engineers are confronted with various alternatives to be considered. In such situations, property refinements can help in exploring alternative proposals and thereby providing the most appropriate abstraction.

For a property-driven requirements engineering, we instantiate our RE methodology in a way that, starting from functional properties described by textual property descriptions, a stepwise formalization of functional requirements via interface assertions is enabled, which are summarized in function specifications. Developing a component architecture and linking the function specifications to this architecture was not part of this case study.

5.4.2 Goal

This case study follows the goal of evaluating the applicability, benefits, and limitations of integrating our model-based RE methodology in a property-driven requirements engineering context.

5.4.3 Study Object

We applied the RE methodology in a one year industrial collaboration with Siemens Rail Automation. The goal of the project was to apply and evaluate a seamless model-based development approach in the context of rail automation. Therefore, we modeled and formalized an extract of a train control system starting from informal natural language requirements to implementation code in a stepwise process and assessed the benefits and efforts with Siemens. The Trainguard MT[5] is a modern modular automatic train control (ATC) system that offers many features, such as train supervision and execution and control of the entire range of railway operations. It furthermore supports different levels of train control automation depending on the characteristics of the train and the track segment. From this system, we decided to develop the platform screen doors function.

The Platform Screen Doors Function The Platform Screen Doors Function (PSD) is one feature of the Trainguard MT system. Its purpose is to supervise passenger exchange at platforms. The supervision of train doors is provided by the feature on each level of automation, such that, for example, it is not possible to activate the propulsion if the train doors are still open. Some platforms are equipped with additional doors mounted at the platform, which only open if there is a train in the station. The purpose of the feature is to monitor and control these platform screen doors (PSDs) to synchronize the opening and closing of train and platform screen doors. Figure 5.32 shows a typical scenario, in which a train enters a station and the train and platform screen doors open and close simultaneously.

[5] http://www.siemens.com/press/pool/de/feature/2013/infrastructure-cities/mobility-logistics/2013-02-trainguardmt/broschuere-trainguard-mt-e.pdf

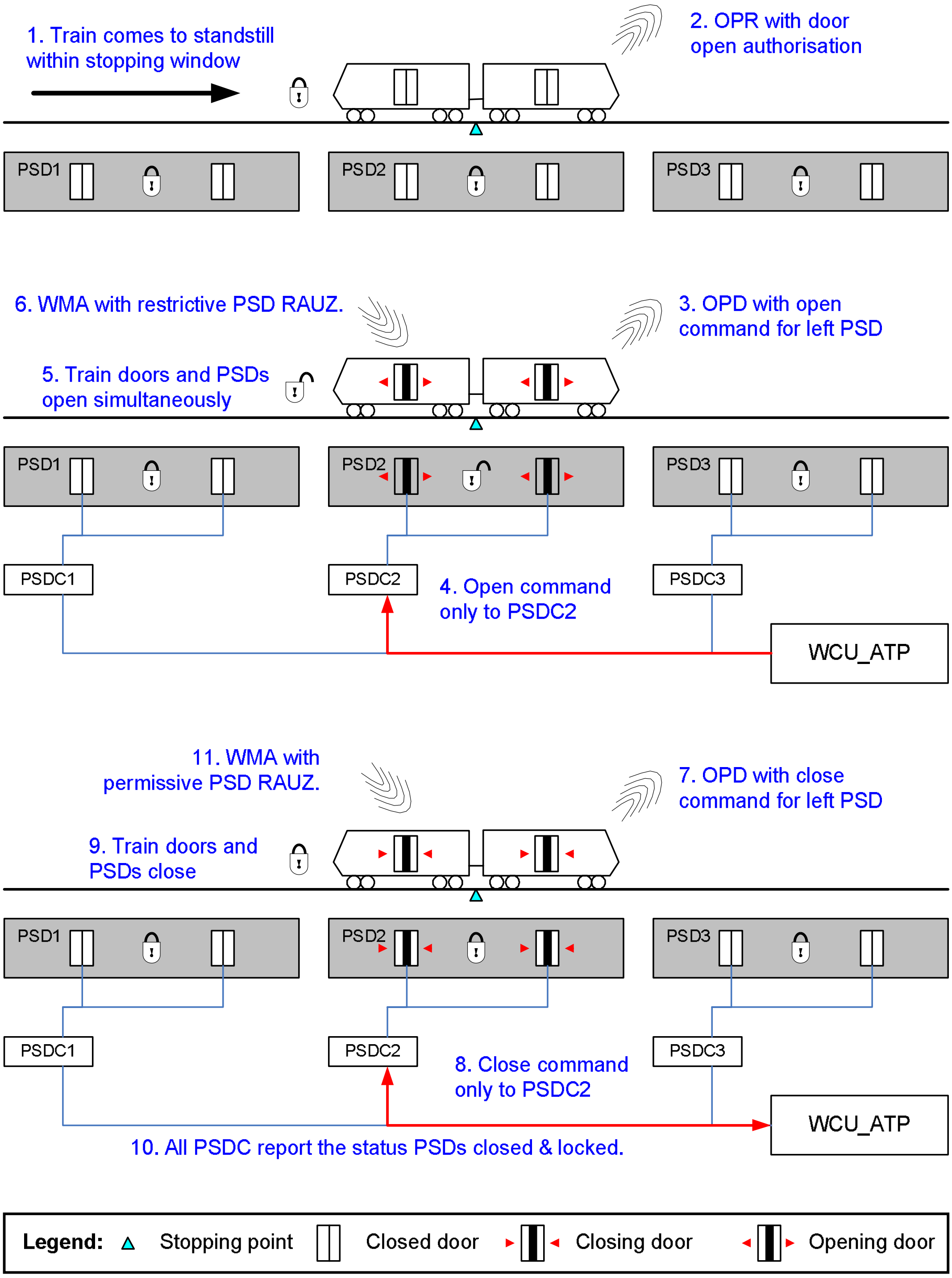

1. Train comes to standstill within stopping window
2. OPR with door open authorisation
PSD1
PSD2
PSD3
6. WMA with restrictive PSD RAUZ.
3. OPD with open command for left PSD
5. Train doors and PSDs open simultaneously
PSDC1
PSDC2
PSDC3
4. Open command only to PSDC2
WCU_ATP
11. WMA with permissive PSD RAUZ.
7. OPD with close command for left PSD
9. Train doors and PSDs close
8. Close command only to PSDC2
10. All PSDC report the status PSDs closed & locked.
Legend:
Stopping point
Closed door
Closing door
Opening door

Figure 5.32: Scenario of platform screen door (PSD) control and monitoring. OPD, WMA and OPR are specific data telegrams for communication between train and platform. RAUZ is short for run authorization zone. The functionality is executed on four controllers (WCU_ATP and PSDC1–3).

5.4.4 Study Execution

As an input for the execution of the case study, Siemens provided two documents: A *System Requirements Specification (SRS)* and a *System Architecture Specification (SAS)*. Both documents contained requirements with the difference that the requirements in the SAS were already structured according to a set of subsystems and the interface descriptions were more concrete and closer to the technical signals used in the implementation.

We applied our RE methodology to the requirements specified in both documents and proceeded to a system specification consisting of a function architecture with associated function specifications and a mode model. The project was conducted over a duration of six months with four researchers working on the model.[6] Within the six months, we had three extensive full-day workshops with experts of Siemens Rail Automation. In the three workshops, we discussed and assessed the quality and correctness of the resulting specification.

5.4.5 Instantiation of the RE Methodology

To apply the artifact model in a property-driven requirements engineering context, we instantiated and tailored the artifact model, and provided a concrete representation. Although the study object description contained one scenario for illustration (Figure 5.32), the requirements given in the input documents were primarily formulated as functional properties the system shall fulfill. In contrast to the scenario-driven approach taken in the last section, the requirements were not given by use case descriptions or possible usage scenarios of the system. Instead, the requirements were structured according to functions the system shall fulfill and stated as independent assertions that correspond to reactions the system shall show, assuming some situation and stimuli given by its environment. The requirements are thus independent from each other to some extent and cannot be connected to a specific usage scenario.[7]

Based on the requirements documents given as input for the case study, we used only a subset of model types from the artifact model described in Section 5.1. The process we took in this case study to create and verify artifacts, is depicted in Figure 5.33. The figure shows the order in which we created artifacts of the artifact model (solid arrows) and checked their consistency (dashed arrows). A detailed description of this study execution process is given in the following.

Steps 1 and 2: We started by deriving an initial *function list* from the input documents that defined the scope of our case study. This initial list contained, for example, the functions *Door Supervision and Control* and *Train Departure Control*. We extracted all functional requirements related to these functions from the SRS and SAS documents and documented them as *textual property descriptions* as depicted in Figure 5.34.

[6] The four researchers were not working full time. The effort spent was roughly 4 PM.

[7] For Siemens, the independence of requirements is a quality criterion.

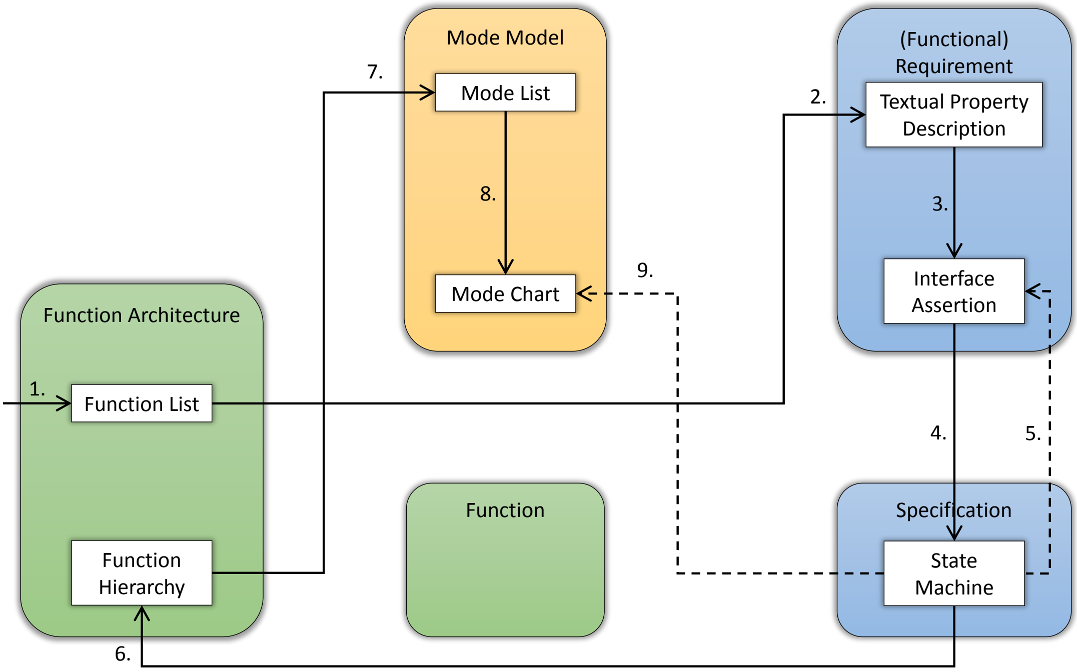
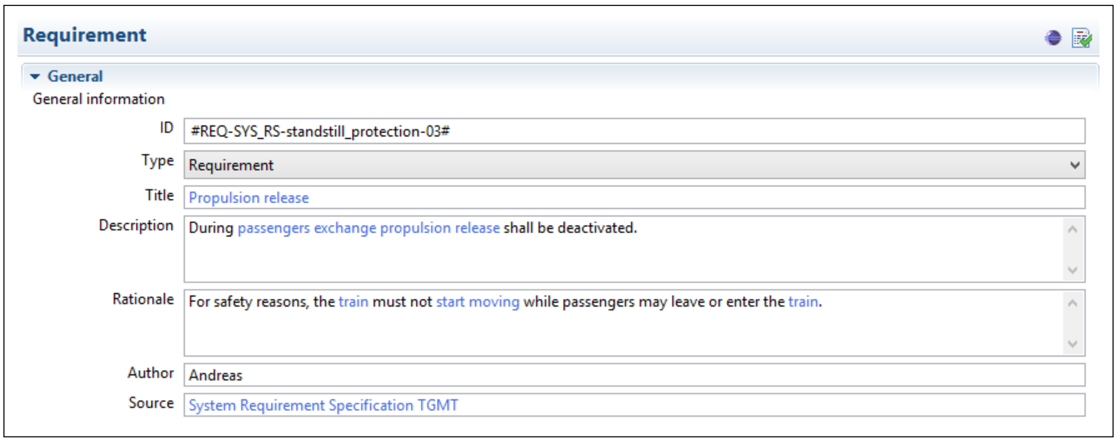


Figure 5.33: Study execution process taken in the property-driven case study.



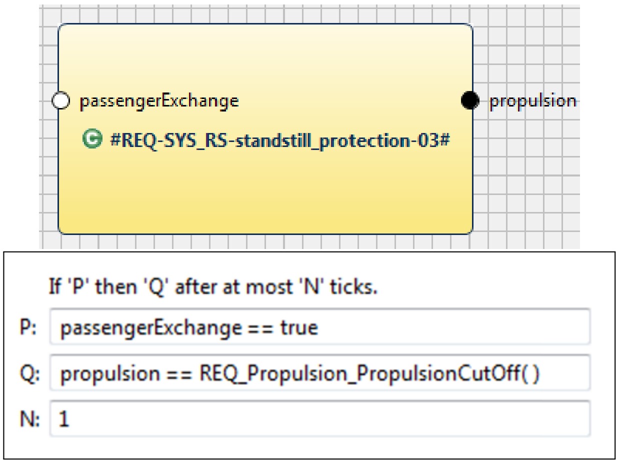
Requirement

General

General information

ID	#REQ-SYS_RS-standstill_protection-03#
Type	Requirement
Title	Propulsion release
Description	During passengers exchange propulsion release shall be deactivated.
Rationale	For safety reasons, the train must not start moving while passengers may leave or enter the train.
Author	Andreas
Source	System Requirement Specification TGMT

Figure 5.34: Example of a *textual property description*.



If 'P' then 'Q' after at most 'N' ticks.

P: passengerExchange == true

Q: propulsion == REQ_Propulsion_PropulsionCutOff( )

N: 1

Figure 5.35: *Interface assertion* with syntactic interface (top) and an associated interface behavior (bottom) formalizing the textual property description shown in Figure 5.34.

Step 3: As third step, we formalized the textual property descriptions by *interface assertions* (see Figure 5.35). In this case study, we adopted the specification patterns of Dwyer et al. [1999] to specify the behavior related to an interface assertion. The result of this process is a set of interface assertions, each having a syntactic interface and an associated behavior.

Step 4: We created *function specifications* by examining the syntactic interfaces of the interface assertions and composing those in one function specification that referenced the same or similar outputs. This integration of interface assertions into function specifications was a design decision made by us and discussed with Siemens. For example, we integrated all interface assertions with outputs related to the opening, closing, and release of train doors to a *Train Door Function* specification. As already described before, a function specification itself is specified by a syntactic interface and an interface behavior. The syntactic interface of the function specification is the result of the composition of the syntactic interfaces of all interface assertions integrated in this function specification. Figure 5.36 illustrates this integration of interface assertions into one function specification. Some of the syntactic interfaces of the interface assertions had inputs that correspond to outputs of other interface assertions. In such cases, the models reflected a situation, where the original textual property descriptions referred to each other (e.g., "If the door mode is *automatic*, the doors shall be opened"). These *internal* references between interface assertions mapped to one function specification represent structural information. We utilized these references to structure the function specifications. Interface assertions that were integrated into one function specification but referenced each other by their syntactic interface, define two separate subfunctions within the function specification. Figure 5.37 shows the decomposition of the *Train Door Function* into three subfunctions as a result of the interface assertions integrated in this function specification. The interface assertions, in this example, reference each other in three cases (*doorMode*, *doorReleaseStatus* and *releaseForced*). These references represent modes of the function specification. As a consequence of this construction approach, we are able to trace and relate each in-

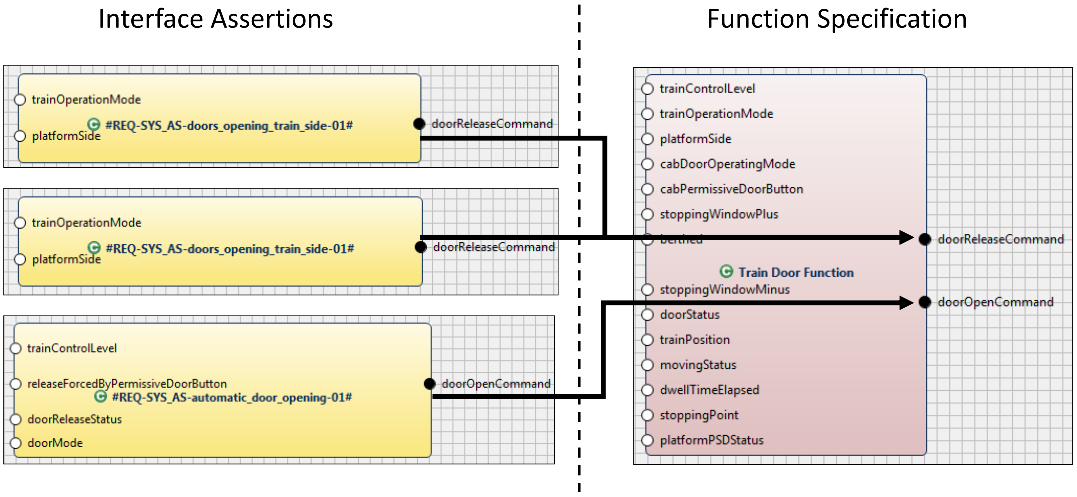


Figure 5.36: Integration of interface assertions into a function specification by composing interface assertions with equal or similar outputs.

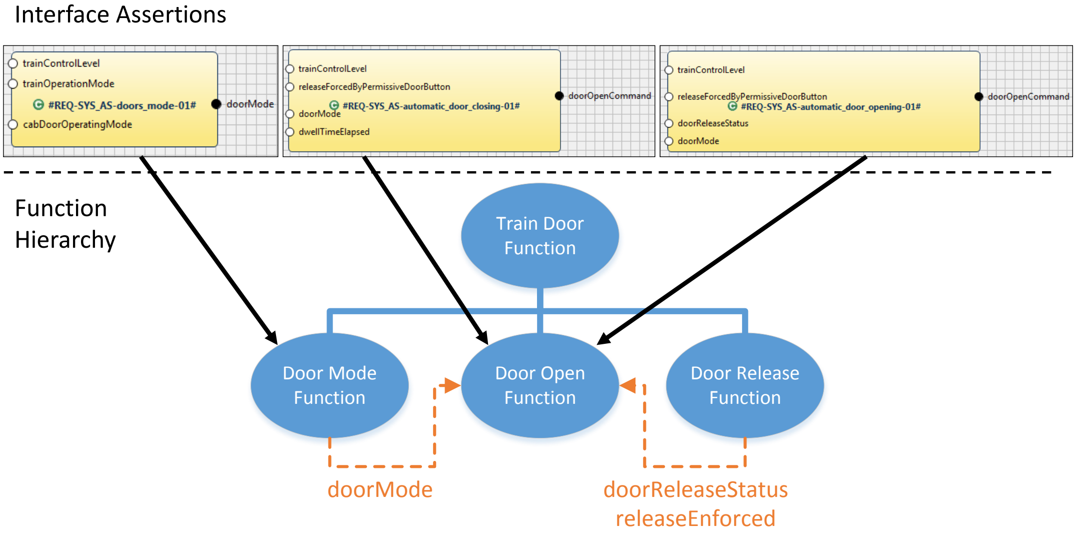
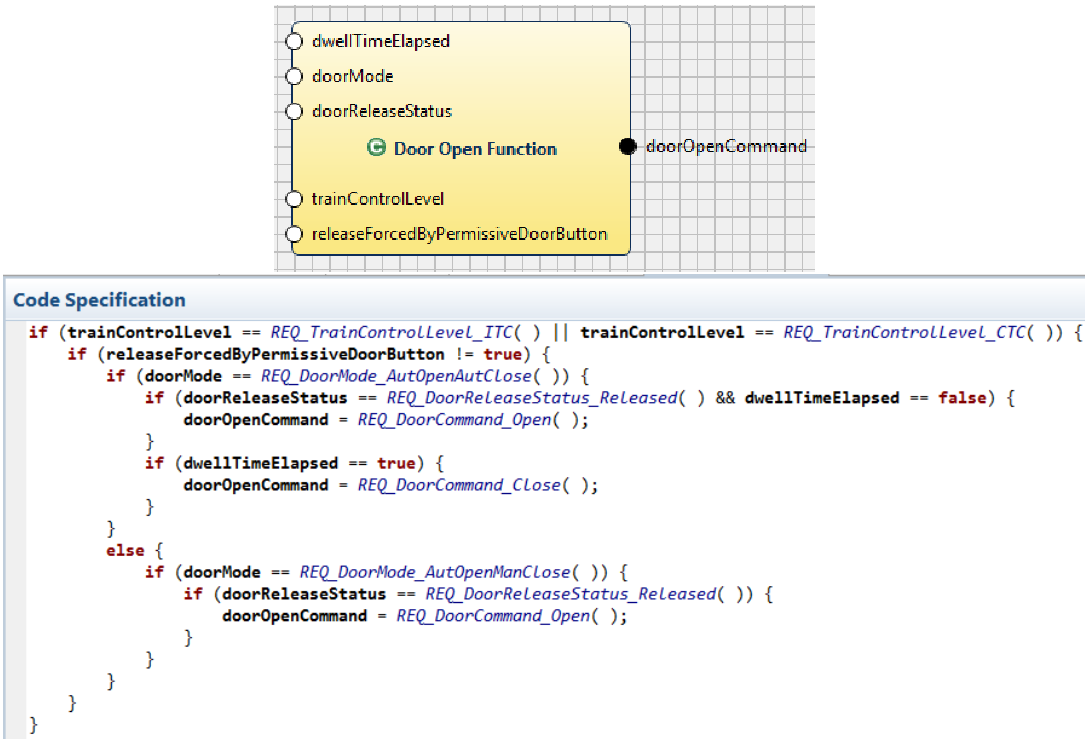


Figure 5.37: Decomposition of the *Train Door Function* into three subfunctions. The three internal signals *doorMode*, *doorReleaseStatus* and *releaseForced* represent modes of the function specification. They originate from the interface assertions (formalized textual property descriptions).



```
if (trainControlLevel == REQ_TrainControlLevel_ITC( ) || trainControlLevel == REQ_TrainControlLevel_CTC( )) {
    if (releaseForcedByPermissiveDoorButton != true) {
        if (doorMode == REQ_DoorMode_AutOpenAutClose( )) {
            if (doorReleaseStatus == REQ_DoorReleaseStatus_Released( ) && dwellTimeElapsed == false) {
                doorOpenCommand = REQ_DoorCommand_Open( );
            }
            if (dwellTimeElapsed == true) {
                doorOpenCommand = REQ_DoorCommand_Close( );
            }
        }
        else {
            if (doorMode == REQ_DoorMode_AutOpenManClose( )) {
                if (doorReleaseStatus == REQ_DoorReleaseStatus_Released( )) {
                    doorOpenCommand = REQ_DoorCommand_Open( );
                }
            }
        }
    }
}
```

Figure 5.38: Example of a *function specification* with syntactic interface (top) and an associated interface behavior described by a code specification (bottom).

terface assertion to exactly one (sub)function specification as depicted in Figure 5.37. The behavior of a function specification was, in the end, modeled by an executable specification technique, such as state machines, table specifications, or code specifications (see Figure 5.38).

Step 5: To check whether the executable function specifications do not violate any of the interface assertions associated with it, we applied model checking to verify the consistency between the function specification and the interface assertions. The compatibility of the syntactic interfaces of interface assertions and function specification that is necessary to perform model checking was ensured by the construction approach described in the previous step.

Step 6: As described in step 4, the function specifications themselves exhibit a subfunction structure that was induced by the interface assertions associated with that function specification. We integrated this structure in a *function hierarchy*. We compared the function specifications with the initial functions from the function list and created a function hierarchy that associated the functions from the function list with the function specifications (and added functions not considered in the initial creation of the function list).

Steps 7 and 8: We investigated the function dependencies between the function specifications associated with the functions of the function hierarchy to derive a mode list for the system. That means, we looked at each mode channel between two function specifications and defined the type of the channel as a mode. The values that were transmitted over this mode channel served as mode values. The result is a *mode list*. From this list, we created a *mode chart* in which we structured the modes as AND or OR modes and added transitions for valid mode switches based on common sense and discussion with the domain experts from Siemens.

Step 9: The mode chart defines a set of valid sequences of mode configurations that must not be violated by the function specifications. Otherwise, the function specifications are not consistent with the underlying mode model, i.e., either a function specification or the mode model contains an error. To check the consistency, the students compared the possible execution sequences of the mode channels in the function specification with the valid sequences of mode configurations from the mode chart.

Missing Model Types: The creation of a component architecture for the system was not in the scope of this case study. Therefore, the model types related to the component architecture are not part of the study execution process. Another interesting aspect of this study execution process is that we did not create any artifacts that describe the purpose and usage context of a function in detail (e.g., by providing a use case table for each function). We did not create these artifacts because the input documents provided no explicit information on this. This was not crucial for the execution of this case study; however, in the next chapter of this thesis, we will discuss the relevance of such high-level descriptions for a comprehensive function documentation.

5.4.6 Study Results

Reflecting the goal of this study to assess the applicability, benefits, and limitations of integrating our model-based RE methodology in a property-driven RE context, we describe our findings structured according to these three subgoals.

Applicability

System Structure Constraints. The methodology and its instantiation as described so far is agnostic with respect to a specific scope chosen to specify, i.e., the methodology is suitable for specifying an entire system (e.g., the whole TGMT system) as well as for specifying a single subsystem of it (e.g., the control system of a single platform screen door).

In this case study, we initially planned to apply the methodology to the scope of the platform screen door function. That means, the scope of the system to be specified is defined by the scope of the TGMT system but only the parts relevant to the PSD function were modeled. However, during the study execution, a specific architecture constraint became so prevalent that we integrated this constraint into the methodology as described in the following.

The essence of the TGMT system lies in the interaction and coordination between the *onboard* system of a train and the *wayside* system of the track. This distinction is so prevalent due to its physical distinction that it can be considered as an architectural constraint rather than a design decision. Thus, we started breaking down the requirements with a scope of the TGMT system to requirements with a scope of the onboard or the wayside subsystem. These requirements were formalized by interface assertions and two function architectures were created, one for the onboard and one for the wayside subsystem.

Granularity of Functional Requirements. System level requirements (from the system requirements specification) and subsystem level requirements (from the system architecture specification) resided on different levels of abstraction. More specifically, they differed in terms of the *scope* that they address and the *granularity* in which they were described.[8]

With respect to the *scope*, the system level requirements describe requirements of the entire TGMT system, whereas the subsystem level requirements refine these system level requirements by breaking them down to requirements for the onboard and wayside subsystems. Thus, subsystem level requirements exhibit a different scope and introduce additional requirements considering the internal communication between the subsystems.

With respect to the *granularity*, the system level requirements describe events more coarse grained than the subsystem level requirements. For example, a system level requirement may refer to the event *Door Open*, while on the subsystem requirements level, this event is described by two events, *Door Release* and *Door Opening*, which describe the abstract event in more detail. This refinement relation can be considered as interaction refinement [Broy, 2010a].

We were able to apply the artifact model on both levels of granularity.

Formalization of Textual Property Descriptions. For the scope of the project, we extracted 49 relevant textual property descriptions from the documents. From these, we were able to formalize 39. In case we were not able to formalize a textual property description, this was because either the requirement did not describe system behavior or the requirement described real-time behavior, which was not in the scope of the case study.

Derivation of Function Specifications from Interface Assertions. The 39 interface assertions were integrated into 9 function specifications (5 for the onboard and 4 for the wayside subsystem). These function specifications were further decomposed into an overall of 13 subfunctions that were specified by behavior specifications (state machines and table specifications).

To a large degree, the function hierarchy could be derived from the interface assertions in a schematic manner as described in Section 5.4.5 (step 4). By grouping interface assertions and identifying internal communication (mode channels), we were able to easily create the structure of the function hierarchy. The resulting function hierarchies for the onboard and wayside subsystems are depicted in Figures 5.39 and 5.40. The function hierarchy of the onboard subsystem shows seven function de-

[8] Teufl et al. [2014] provide an interesting discussion about the implications of this gap (scope and granularity) between requirements.

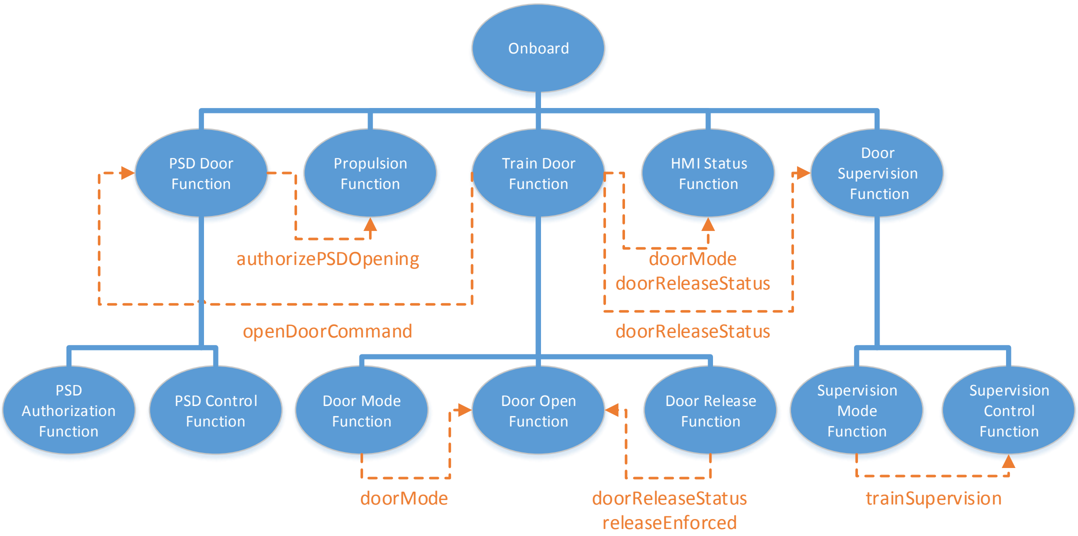


Figure 5.39: Function hierarchy of the onboard subsystem.

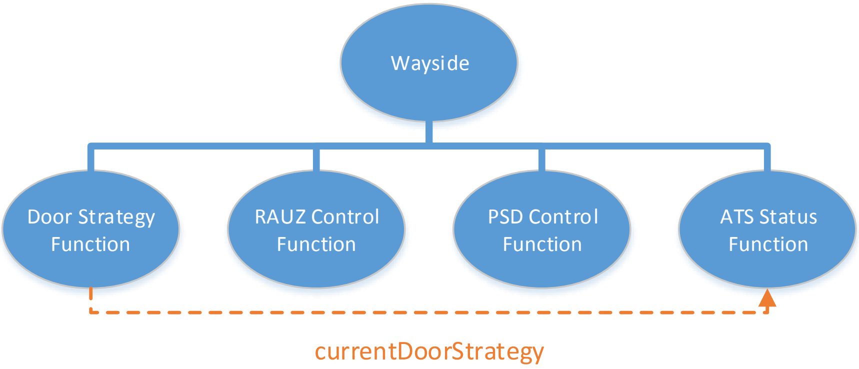


Figure 5.40: Function hierarchy of the wayside subsystem.

pendencies between the functions, whereas the functions of the wayside subsystem only exhibit one function dependency.

Figure 5.41 shows the mode chart of the onboard subsystem that we derived from the function dependencies in the function hierarchy. This representation was well suited due to its concepts of hierarchically nested states and orthogonal regions. Hierarchically nested states are used to keep the diagram comprehensible by grouping a set of substates to a more abstract superstate (e.g., in the figure, the superstate *Operation Modes* actually contains five sub states structured in two orthogonal regions *Release Options* and *Door Mode*). Orthogonal regions are used to model independent modes. Independent modes exhibit an *AND*-semantics, which means that for each region, exactly one state is active at a time (e.g., in the figure, the status of the *Train Doors* is independent from the status of the *Train Supervision* mode).

Formal Verification. Out of the 39 interface assertions, we were able to verify for 31 (ca. 80%) that they are not violated by the associated function specification. In cases, in which the verification failed, this was due to incompleteness or inconsistency of

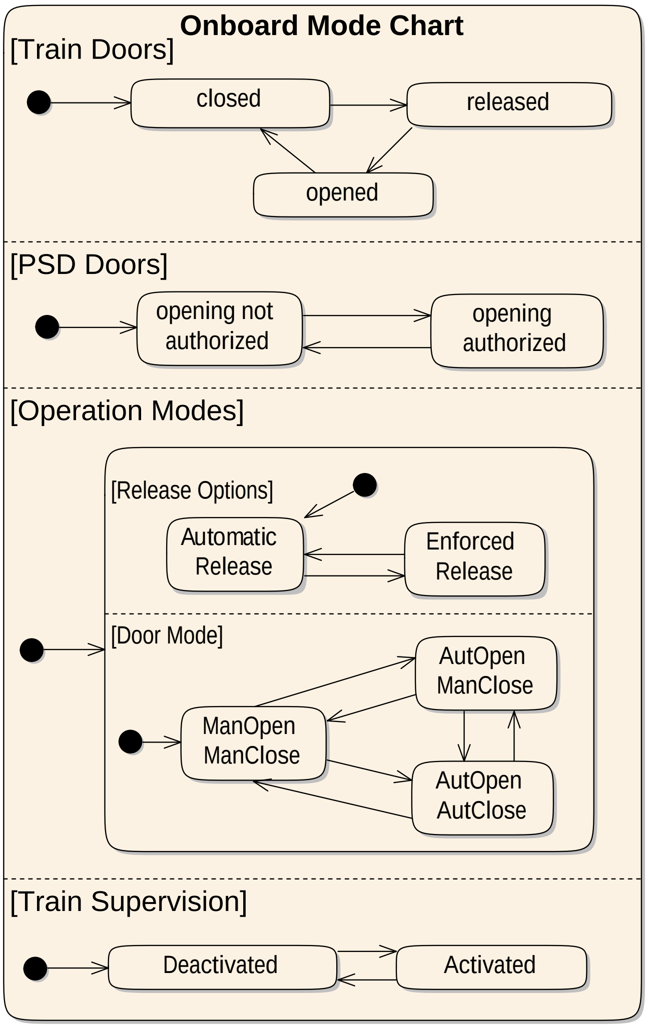


Figure 5.41: Mode chart of the onboard subsystem.

the original requirements provided by Siemens, or due to disability of the tool we used for this case study.[9]

Benefits

Resolved Inconsistencies. Through the formalization and verification of requirements, we revealed requirements that, at least, needed further discussion with the different stakeholders. We found that some requirements lacked information or assumed certain properties of the context. For example, conflicting requirements of one function were revealed when in the verification process either one interface assertion was fulfilled or the other but never both. In these cases, additional information or conditions must be added to the requirements such that the verification process confirms that a function fulfills both interface assertions.

Requirements Tracing. As mentioned before, we applied the artifact model to requirements on different levels of granularity and scope (system level requirements vs. subsystem level requirements). The formalization of the requirements as described in the artifact model enabled defining a (formal) refinement relation between these requirements. Therefore, we were able to verify that the function hierarchies, which were specified on the subsystem level, are consistent with the subsystem level interface assertions and afterwards verify that the subsystem level interface assertions imply the system level interface assertions.

Limitations

Modeling Efforts. In the context of this case study, we were not able to obtain realistic effort estimations. This was because in the project the modeling work was done by the tooling and method experts (academic partner) and not by the domain experts (industrial partner). On the other hand, letting the domain experts do the modeling would also not have produced realistic assessments about modeling efforts because, in a realistic context, developers would have been trained beforehand in using the tools and methodology.

Missing Guidance for Appropriate Levels of Granularity. As described in Section 5.4.6, the requirements of the documents provided by Siemens were on different levels of granularity. This is often the case in development projects, where there is a *requirements specification* and a *system-* or *architecture specification*. The artifact model of our methodology does not prescribe any level of granularity on which requirements or functions *should* be described. It was, for example, not clear whether we should specify the syntactic interface and the associated interface behavior of a function specification with very technical input and output messages resembling the architecture specification (e.g., *TCL_O_Door_Release*) or on a more abstract and intuitive level resembling the requirements specification (e.g., *doorReleaseCommand*).

[9] Tool support is discussed separately at the end of this chapter.

5.4.7 Threats to Validity

Since the study object of this case study is a system that is already implemented and introduced in several markets, the requirements specification provided by Siemens evolved and was influenced by the implementation and its architecture. This poses a threat to the *internal validity* of this case study since the high applicability of our methodology may be based on the requirements documents being already aligned to a specific architecture. Additionally, the modeling of the system was not done by the domain experts but by the experts of the methodology. This may affect the results with respect to the applicability of the methodology. We tried to mitigate this threat by demonstrating and assessing our proceeding with the domain experts in regular workshops.

As this was a first exploratory feasibility study in the context of a property-driven RE context applied in the rail automation domain, we cannot guarantee that the results are generalizable to other systems from the automation domain or even property-driven RE context in general.

5.5 Tool Support

In both case studies, we used the tool AutoFocus3 (AF3)[10] to implement the entire methodology. Requirements, function hierarchies, mode models, architecture, and even code generation of both case studies were modeled in AF3. The figures used in Sections 5.3 and 5.4 show screenshots from the tool.

AF3 is originally a CASE tool for the model-based development of embedded systems. It supports the creation of hierarchical, component-based software and hardware architectures. Components in the architecture hierarchy can be equipped with behavior specifications, for example, using state machines, I/O tables, or code specifications.

Recently, AF3 has been extended by a requirements module called MIRA [Teufl et al., 2013]. With MIRA, requirements engineers can document and formalize requirements directly in AF3 and link them in various ways to the architecture. One form of requirements that is supported by MIRA is use case tables with scenario descriptions. The scenario descriptions can be formalized using MSCs. Furthermore, formal specifications for requirements can be developed using the same modeling techniques as for the architecture.

Furthermore, AF3 supports all types of analyses mentioned in the case study sections out of the box (e.g., MSC feasibility checks, refinement tests, and model checking of temporal logic assertions).

5.6 Discussion

In the two exploratory case studies, we have seen that the proposed artifact model is applicable to different RE approaches and application domains. The results in-

[10] `http://af3.fortiss.org`

dicate that the application of the artifact model facilitates the revelation of implicit function dependencies within the requirements and the explicit documentation of them. However, the results also indicate that further work is necessary to address the mentioned limitations experienced in the case studies, which are discussed in the following.

One issue that occurred in the case studies was the choice of an appropriate level of granularity on which the artifact model should be applied. Actually, in the property-driven RE context, we applied the artifact model on two different levels of granularity (see Section 5.4.6). This decision was motivated by the different levels of requirements granularity handled in the two input documents provided by Siemens. However, it remains an open question whether these two levels are appropriate to support development activities, and what these development activities actually are. This calls for a study on development activities that a function specification should support and the appropriate level(s) of granularity on which the artifact model should be applied.

A second point for discussion is the missing guidance for designing a function, which includes the decomposition of functions into subfunctions and the specification of their behavior by means of behavior specification techniques (e.g., state machines). The artifact model itself does not give any guidance for that. In the case studies, we decided case by case whether and how a function is decomposed into subfunctions and which specification technique to apply. An open question is whether this process can be guided to a certain extent, assuming that there are reoccurring types of functional intent that can be handled in a similar way. This resembles the idea of *design patterns* [Gamma et al., 1994] for function specifications.

The two issues mentioned above lead to the question, whether we can provide a template for a comprehensive *function documentation* that integrates all relevant information about a function, and what these information are. In the following chapter of this thesis, we pick up this question and provide a solution.

Another point that needs further examination is the elicitation and derivation of a mode model. In the two case studies, the modes were defined in an *ad hoc* manner. That means, in the first case study, we defined modes based on a document inspection and then added modes, whenever we needed them for the modeling of a function dependency. In the second case study, we derived the modes by their implicit use in the textual requirements of the input documents. These procedures were sufficient for describing the function dependencies in the case studies; however, we think that a complete and consistent mode model has a positive impact also on the development of new functions and the evolution of existing ones. Systematic elicitation approaches are necessary to reach these goals. In Chapter 7, we introduce and assess three systematic elicitation approaches for mode models.

5.7 Summary

In this chapter, we presented the integration of a formal specification framework for multifunctional systems into a model-based requirements engineering methodology. We defined the methodology by describing an artifact model that relates the semantic concepts of the formal specification framework to model types that we use to describe them. The relations between the semantic concepts are reflected and expressed by the model types. We highlighted the role of the resulting system specification in a development process and showed its potential for automation.

We presented two case studies, where we applied the methodology in a scenario-driven requirements engineering context and in a property-driven requirements engineering context. The case studies showed that the methodology is feasible and allows detecting inconsistencies in requirements early, especially with respect to function dependencies.

The discussion of the case study results revealed the need for additional methodical guidance especially with respect to the level of granularity on which the artifact model should be applied and the decomposition and specification of functions. Additionally, more work is needed on a systematic elicitation of mode models. Both issues are subject to the following chapters of this thesis.

Chapter 6

Function Documentations for Multifunctional Systems

The application of the artifact model as described in the previous chapter revealed missing methodical guidance when it comes to the selection of an appropriate level of abstraction for describing functions as well as the design of a proper function specification. In the case studies presented in the last chapter, we decided case by case whether and how a function is decomposed into subfunctions and which specification technique to apply. If we want to introduce a comprehensive integration of our specification approach into the requirements engineering for multifunctional systems, we must provide additional guidance on how to specify and document functions. This leads to the question, whether we can provide a template for a comprehensive *function documentation* that provides all relevant information about a function, and what these information are.

Giving specific advice on how to create a function documentation calls for an in-depth study on working practices and impediments in a community of people working with function documentations. In this chapter, we present such a study revealing activities performed on the basis of function documentations, challenges faced when working with them and relations between these challenges. The conclusions of this study particularly address the way in which function documentations should be structured and which information they should contain. From the results of the study, we develop a new structure for function documentations that describes a function on *three levels of abstraction* and structures it by *modes of operation*. The resulting function documentation template provides methodical guidance for the creation of a function documentation in the context of our RE methodology.

This chapter is partly based on a previous publication [Vogelsang, 2014].

6.1 Study Context: Function Documentations for Automotive Systems

Function documentations are central artifacts in the development of automotive systems. They describe the requirements and the basic design of a *function*, which summarizes a specific part of functionality a vehicle exhibits. Function documentations are used for several purposes, e.g., documenting requirements, communication between OEM and supplier, calibration of system parameters in a specific product, or testing and implementing a function. This function documentation artifact in the automotive industry incorporates information that can be linked to several model types introduced in the artifact model of our RE methodology (see Section 5.1).

Over the last 25 years, the number of functions (and thus the number of function documentation artifacts) in a vehicle increased and the interaction and logical dependencies between functions became more relevant for their specification and documentation [Vogelsang and Fuhrmann, 2013].

As a consequence, developers and testers have difficulties to comprehend the complex function documentations, to distinguish between requirements, which need to be fulfilled and design decisions, which can be altered, and to assess the impact of changes [Broy, 2006]. However, it is unclear, what the reasons for these difficulties are. What are the challenges the developers and testers face? How are these challenges related? How can we improve function documentations and which information must they contain to support development activities in a situation, where function complexity and coupling increases?

In different contexts and companies, function documentation artifacts are called *function definition*, *function specification*, *functional specification*, *function requirements documentation*, or *software and calibration documentation*. This artifact is a written documentation of a function's requirements and its abstract design. In many automotive companies, models such as dataflow specifications are used to augment textual descriptions (cf. [Robinson-Mallett, 2012]). These dataflow specifications may be either used to generate code for the function (e.g., by using Matlab Simulink or ASCET models) or the models just serve as an overview and the function is implemented separately (e.g., as plain C code).

The special role of function documentations in the automotive sector originates from the fact that function documentations are not solely a documentation for function developers. They are the basis for a number of activities and they are read by function developers, system integrators, functions testers, and calibration engineers.

We conducted the study that we will present in this chapter at Robert Bosch GmbH, a worldwide supplier of hardware and software for automotive systems. At Bosch, function documentation artifacts are textual documents that are structured into four main sections described in the following. The structure and content is similar to function documentations we have seen in other automotive companies. The names of the sections are taken from the original documents and their appropriateness may be subject to discussion. In the context of our study, the actual section names are not important.

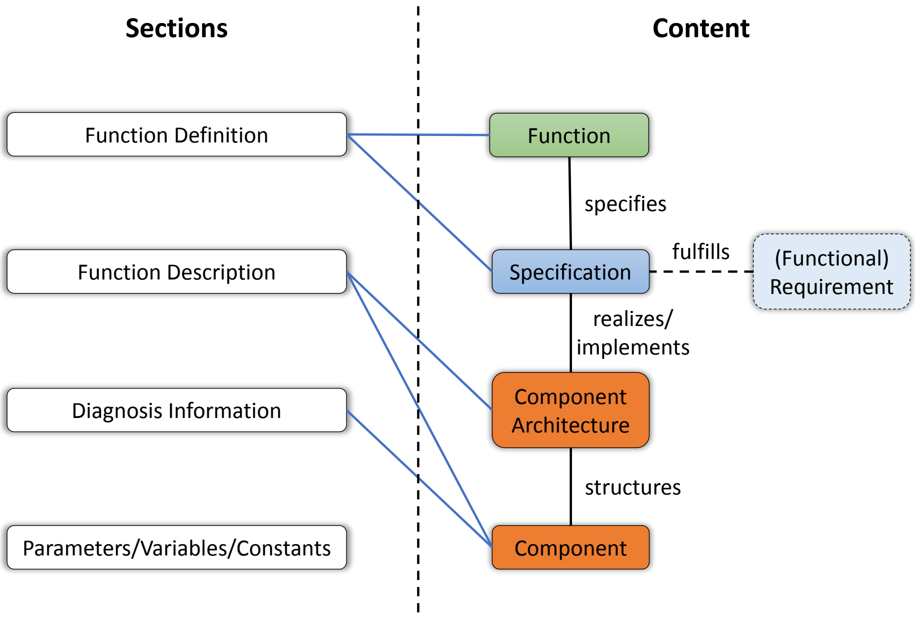


Figure 6.1: Sections of current function documentations related to the content they describe.

Function Definition: The *function definition* section contains an abstract description of the task that the function is supposed to realize. This description is often given textually. In addition to this textual high-level description, the section also contains a graphical representation of the black-box interface of the function. This black-box interface contains the input signals that the function uses and the output signals that the function produces without giving any details on its internal structure.

Function Description: In the *function description* section, the internal structure and realization of the function is described by hierarchical dataflow specifications and accompanying textual descriptions. The function description section is structured according to the hierarchy of the dataflow specification (cf. [Robinson-Mallett, 2012]).

Diagnosis Information: The *diagnosis information* section contains information about the monitoring of erroneous situations within the function. The representation is similar to that of the function description.

Parameters/Variables/Constants: The *Parameters/variables/constants* section contains tables for parameters, variables, and constants that are used in the function documentation. The section summarizes them and provides additional information such as a detailed description of a signal and its type.

We can classify the information provided in these sections with respect to our semantic model introduced in Chapter 2. Figure 6.1 shows the relation between the sections and the content they express. We see that a large part of the function documentation is concerned with the realization of the function. Functional requirements do not have a dedicated (sub)section. They are, if at all, documented as textual explanations attached to the function specification. The function specification content focuses on the description of the syntactic interface in terms of input and output signals.

From a series of collaborative projects with a number of automotive companies in the last years, we made the following observations concerning the current state of function documentations in practice:

- Function documentations are merely a documentation of the implementation and do not describe the requirements explicitly.
- Function documentations are hard to comprehend for people who are not involved in the development of these functions.
- Function documentations are not complete, in the sense that some details that are necessary to understand the function are only present in the source code.

We are interested in exploring and understanding the reasons for these observations. From this understanding, we expect to get insights that help us to provide methodical guidance and best practices for structuring function documentations and to understand which content may be missing or obsolete.

6.2 Research Objective

The purpose of this study is to analyze *function documentations* for the purpose of *identifying and relating challenges* with respect to *comprehensibility* from the point of view of *function developers, system integrators, function testers, and calibration engineers* in the context of an *automotive software company*.

6.3 Study Design

We performed an exploratory study using a grounded theory approach [Adolph et al., 2011] to conduct and analyze semistructured interviews (lasting 1–2 hours) with nine practitioners from Bosch regarding their working practices and experiences with function documentations. The study was led by the following research questions.

6.3.1 Research Questions

RQ1: Which activities are performed on the basis of function documentations? We are interested in process steps and activities that use the function documentation as input or output artifact. We additionally want to assess, which parts of a function documentation are of special interest for which activities. This research question contributes to the goal of exploring the usage and requirements for function documentations.

RQ2: What are the main challenges experienced when working with function documentations? Where are problems and shortcomings when working with function documentations? What might be reasons for them? This research question contributes to the goal of improving function documentations for different stakeholders.

RQ3: How are the challenges related? Can the challenges be related in a way that they result in a comprehensive problem description? This research question contributes to the goal of understanding the reasons for problems in comprehending and documenting a function.

6.3.2 Research Method

To answer the research questions, we followed a *grounded theory* (GT) approach, an exploratory research method originating from the social sciences, becoming increasingly popular in software engineering research [Adolph et al., 2011]. GT is an inductive approach, in which interviews are analyzed in order to derive a theory. It aims at discovering new perspectives and insights, rather than confirming existing ones. As part of GT, each interview transcript was analyzed through a process of *coding*: breaking up the interviews into smaller coherent units (sentences or paragraphs), and adding *codes* (representing key characteristics) to these units. We organized codes into *categories*. To develop codes, we applied *memoing*: the process of writing down narratives explaining the ideas of the evolving theory. When interviewees progressively provided answers similar to earlier ones, a state of saturation was reached, and we adjusted the interview guidelines to elaborate on other topics (cf. [Greiler et al., 2012]).

We applied this research method at a department of Bosch that is responsible for the research and development of software-intensive vehicle systems. The department has longtime experiences in the development of automotive systems and continuously improves its development methods and techniques. The development processes and techniques within Bosch vary depending on the application domain and range from full-fledged model-based approaches with code generation to conventional code-based development. Requirements and specifications for all systems are documented in function documentation artifacts. Systems are integrated based on a signal database, which consists of all signals exchanged over a bus system or within an electronic control unit (ECU). Bosch is responsible for the entire development process of single systems and functions starting from specifying requirements over system design, implementation, testing, and calibration for existing vehicles.

6.3.3 Participant Selection

For the interviews, we selected professionals with experience in working with function documentations between 5 and 10 years. We selected the participants based on their availability and experience and with the goal to cover as many roles as possible in order to explore the whole spectrum of activities performed with function documentations. Table 6.1 lists the participants and their roles.

System integrators are responsible for the integration of several functions into a system. *Function testers* check the correctness of function implementations according to the function specification. *Function developers* develop a solution concept for a function based on its high-level requirements and implement this concept. *Calibration engineers* configure a function for a specific product by fixing its parameters.

Table 6.1: List of roles and corresponding participants.

Role	Participants
System integrator	P1, P9
Function tester	P2, P3
Function developer	P4, P5, P7
Calibration engineer	P6, P8

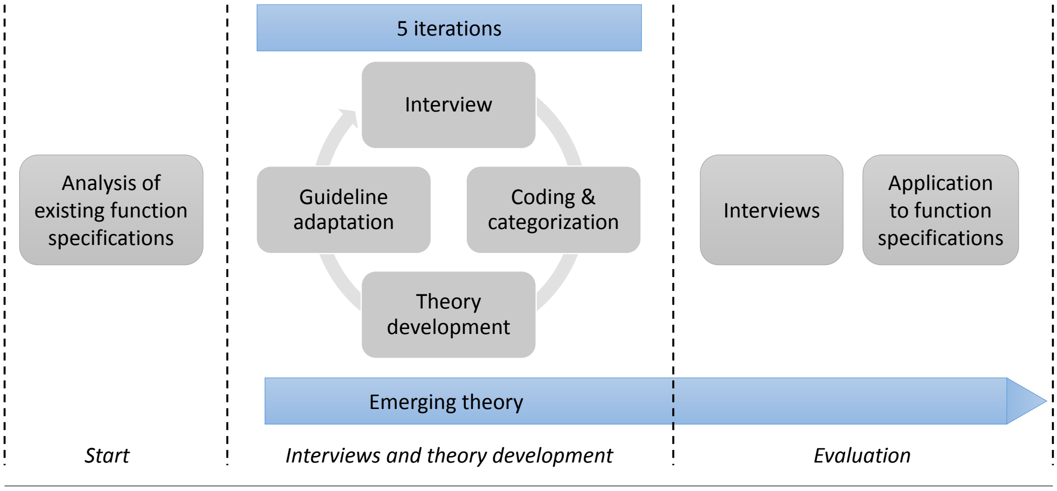


Figure 6.2: Phases of the study [Greiler et al., 2011].

6.3.4 Study Execution

We conducted the study in three phases that are illustrated in Figure 6.2 and described in the following.

Start To familiarize ourselves with the development process and terminology used at Bosch, we analyzed two exemplary function documentations from the engine control domain. To discuss relevant aspects in a more concrete way, we additionally handed out these examples of function documentations in the conducted interviews.

Interviews and theory development In the second phase of the study we conducted five interviews following a rough interview guideline with seven participants overall (one of the interviews had three participants). After each *interview*, the interview transcript was subject to a *coding and categorization* process. The codes were afterwards integrated and related to an emerging theory describing how developers work with function documentation and which impediments they experience. If necessary, the guideline was adapted for the next interview with respect to the evolving theory and possible new insights. In the following, the different steps of this phase are described in detail.

Data collection interview: The interviews followed an interview guideline to cover all relevant aspects. The interview guideline contained open questions about activities performed with function documentations and their role in the development process, used terminology, importance of specific content for an activity, related artifacts, missing information, and comprehensibility of function documentations. The interviews lasted between 1 and 2 hours and a written transcript was created for each interview.

Coding and Categorization: We broke up the interview transcript into smaller coherent units (sentences or paragraphs), and added *codes* (representing key characteristics) to these units (*open coding*). We afterwards grouped the codes into categories. We created the categories such that they reflect activities that are performed on the basis of a function documentation.

Theory development: Based on the codes and categories, we started looking for relationship between them by connecting them (*axial coding*). A relationship can, for example, be a causal condition (what influences a code or category), a strategy (for addressing a code), or a consequence (result from a strategy) (cf. [Corbin and Strauss, 2008]). We annotated these relations with narrative explanations (memos), which we discussed and evaluated with the practitioners in the following interviews. The emerging theory consists of a set of codes grouped into categories representing activities and relations between the codes.

Guideline adaptation: As the theory emerged during the interviews, we adapted the guidelines to explore and validate new insights and hypotheses originating from the theory.

The data analysis process consisted of a *constant comparison* of text units, codes, and categories, i.e., data was collected and analyzed simultaneously [Adolph et al., 2011].

Evaluation The theory development process of the second phase resulted in a set of hypotheses and explanations about which content of function documentations is important for which activity, where the developers encounter impediments and how these relate to each other. In the third phase of the study, we focused on the most frequent codes and categories and explored them in more detail to refine and evaluate related hypotheses and to derive implications for improvements. In the following, the different steps of this phase are described in detail.

Evaluation interviews: In the third phase, we conducted two interviews, where we presented the theory developed in the second study phase together with the hypotheses concerning the most prevalent codes and categories. We asked the interviewees to assess the validity of our theory and hypotheses and explain the relations between the core concepts in more detail (*selective coding*). A written transcript was created for each interview.

Application: Besides the qualitative evaluation of the theory in the phase 3 interviews, we additionally developed an improved function documentation structure based on the results of the developed theory. This improved documentation structure will be introduced later in this chapter. We applied the new documentation structure to the functions explored in the first phase of the study and assessed it with respect to feasibility and improvements. The resulting new documentations were also discussed in the interviews of the third phase.

6.3.5 Data Analysis

The result of our study is a *theory* that is based on a set of codes grouped into categories representing activities and relations between the codes. Codes represent key characteristics of one or more coherent units (sentences or paragraphs) of an interview transcript. From the codes, we created categories that reflect activities or process steps, and grouped the codes in these categories. Additionally, we connected codes to each other if we found a relationship between them (e.g., one is a causal condition for, a strategy for, or a consequence of the other).

From this network of codes, categories, and relations, we derive hypotheses by which we try to answer the research questions. We answered RQ1 (*Which activities are performed?*) by listing and ordering the created categories by the number of interview appearances of codes related to them. This shall give an overview and a prioritization of activities that need to be supported by a function documentation. For RQ2 (*What are the main challenges?*), we list and investigate the five most frequently assigned codes that appeared in at least half of the interviews. This shall point to the *hot spots* of challenges that need methodical guidance. For RQ3 (*How are the challenges related?*), we traverse the three longest paths in our network of related codes. Together with the memos, we added to a connection of two codes, we derive three sequences of related codes and memos that represent conclusive arguments (*stories*) of the theory. These stories shall serve as rationales for the methodical guidance and improvements, which we derive from the study results and which we describe at the end of this chapter.

6.4 Study Results

In the seven interviews, we applied 47 different codes in total. From these codes, we created eight categories and grouped the codes according to them.

6.4.1 Performed Activities (RQ1)

As mentioned before, we created the categories according to activities or processes that are performed on the basis of function documentations. Table 6.2 shows the eight categories and ranks them by the total number of appearances of codes related to the category. Each code was related to exactly one category.

Comprehending a function According to this ranking, the *comprehension of a function* was the activity that is addressed most in the interviews. 13 codes are assigned to this category, which were added 46 times to units of the interviews. Especially in the interviews with function developers and calibration engineers, the comprehension of a function accounted for a large part of the codes applied to the units of the interviews. P4 states, *"I have to comprehend what a function does"* and *"The demand is to*

Table 6.2: Categories of the theory ranked by the total number of appearances of codes related to the category. The middle column indicates the number of codes related to a category.

Category	# codes	# code appearances
Comprehending a function	13	46
Illustrating the principle of operation	6	29
Documenting requirements	4	22
Structuring a function	5	21
Tracing requirements	3	14
Testing a function	7	13
Calibrating a function	4	13
Creating a specification	5	8

describe what *a function does and not* how". P7 says, *"I first look at the textual description of what the function does, and then at the overview diagram (basic interface, input/output). Then you have to go into details, depending on what you're looking for."*

Illustrating the principle of operation The second most frequently mentioned activity is *illustrating the principle of operation* of a function with 29 appearances of codes related to this category. By this activity, the interviewees referred to the necessity to illustrate a function at a high level of abstraction without any details of the implementation. P4 says, *"A physical description of a function is more useful in the beginning."* and P6 states, *"For an initial understanding, I refer to keywords or names of components. For technical details I might as well just read the code."* Some interviewees also said that they try to figure out what the primary physical inputs and outputs of the functions are and how they are related to each other. P2 proposes, *"I want to describe patterns of behavior for the primary input and output measures of a function."*

Documenting requirements 22 codes were assigned to the activity *documenting requirements*. This activity shows the importance of the function documentation artifact as a communication means for changing requirements. P5 explains, *"Change requests addressing the function documentation are the real requirements. If these are incorrectly documented, something wrong gets implemented."* Function documentations are also used for clarifying abstract requirements with the customer. P1 says, *"In the beginning, requirements are very abstract. The function developer writes a first draft of the function specification that is discussed with the customer."*

Structuring a function To support the comprehension of a function, *structuring a function* was considered important. In the interviews, two main structuring principles were mentioned for functions: hierarchies and modes. P5 mentions, *"Function specifications are hierarchically structured and may also be embedded in a system specification containing also other function specifications."* Modes were mentioned as suitable structuring principle for functions running through a set of phases. P4 explains, *"If there are logically independent parts within a function, a mode-based structure is better than a purely dataflow-oriented model of the function."*

Others *Tracing requirements*, *Testing a function* and *Calibrating a function* are additional activities that are performed on the basis of the function documentation artifact. Codes related to the *creation of a function documentation* were only assigned to 6 units in the interviews.

Use of Information in Function Documentations

In addition to the activities that are performed on the basis of the function documentation artifact, we were interested in the information that is used to perform these activities. Therefore, we asked the participants of the interviews, which parts of the function documentation they use to perform the activities.

Figure 6.3 illustrates the mean value that the participants selected on a Likert scale with five possible values from "I use this section very intensely" to "I don't use this section at all". The sections relate to the structure as introduced in Section 6.1. The results show that for all activities the content of the *Function Description* section is used most intensively. For the comprehension of a function and the illustration of the basic principle of operation, the high-level description in the *Function Definition* section is also very important, whereas, when it comes to more concrete tasks, such as tracing, testing, and calibration, the content from the *Diagnosis Information* section becomes more important. This section contains detailed information especially about exceptional situations and border cases.

As a side question, we additionally assessed the use of textual and graphical representations within the document. Figure 6.4 illustrates the mean value that the participants selected on a Likert scale with five possible values from "I use content with this representation very intensely" to "I do not use content with this representation at all". The results show that graphical representations are preferred for all activities, especially for the tracing of requirements.

6.4.2 Main Challenges (RQ2)

In order to name the main challenges faced, when working with function documentations, we list the five most frequently assigned codes that appeared in at least half of the interviews. Table 6.3 shows the results of this list.

Highlight the basic principle of operation The most frequently assigned code is *highlight the basic principle of operation*. It describes the necessity to understand the (physical) concept that the software function needs to support (P2, P4, P6, P7, P8).

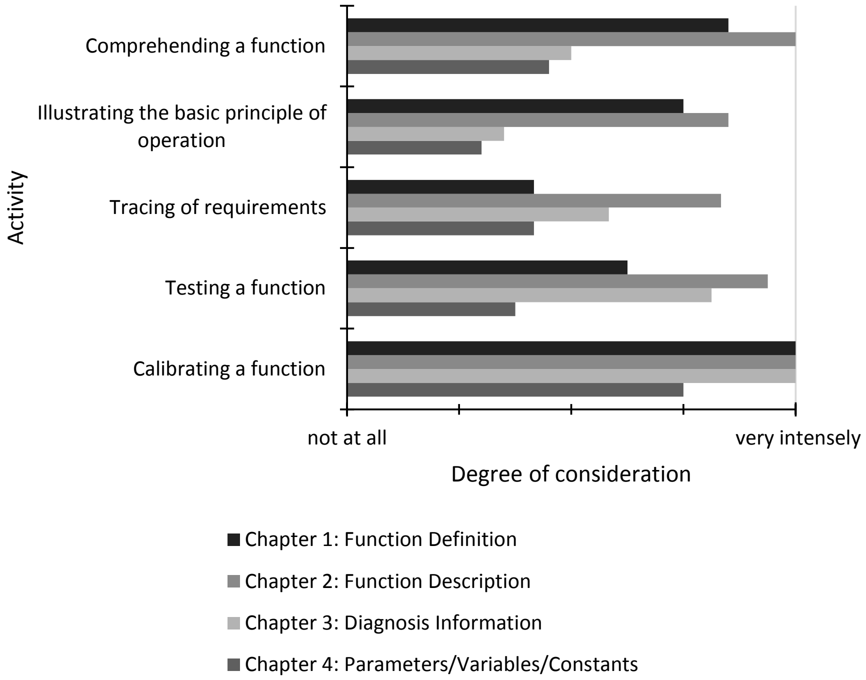


Figure 6.3: Use of document chapters for different activities.

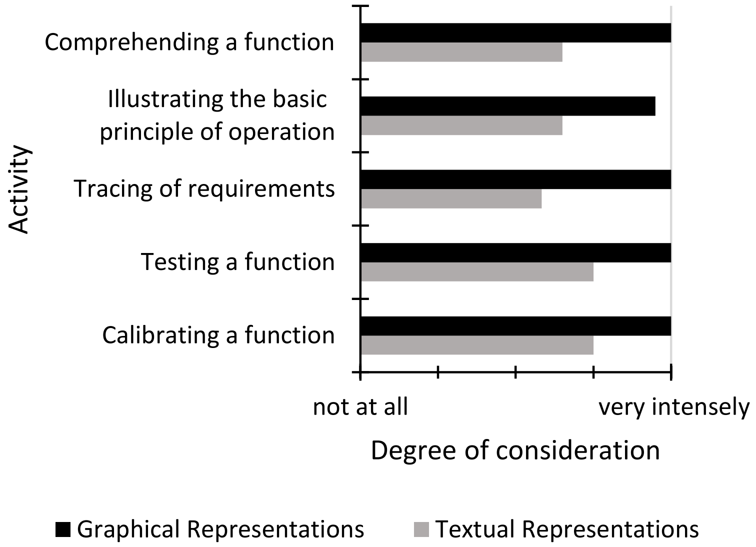


Figure 6.4: Use of content representation for different activities.

Table 6.3: Top 5 codes with appearances in at least half of the interviews. The second column indicates the total number of appearances over all interviews.

Code	Category	#
Highlight the basic principle of operation	Illustrating the principle of operation	15
High-level description of a function	Comprehending a function	8
Deriving requirements from change requests	Documenting requirements	8
Hierarchical structures	Structuring a function	6
Structuring modes and phases	Structuring a function	5

P4 says, *"At first, we focus on the physical aspects of the function and the impact to its operational context. States and implementation details of the software come later."*

High-level description of a function A *high-level description of a function* was mentioned eight times (P2, P3, P4, P5, P6). This code was applied mainly in the context of comprehending a function, especially in the context of comprehending a function someone is not familiar with. P2 claims, *"It is possible to concisely describe the purpose of any function in five sentences."*

Deriving requirements from change requests Third follows *deriving requirements from change requests* (P1, P3, P4, P5, P7). This code addresses the fact that most functionality is not developed from scratch but is driven by change requests applied to older versions of a function. P3 explains, *"We come from initially assuming that there is already an existing solution related to this kind of problem. In many cases we then employ the code of the existing version as the specification for the new version."* Additionally, due to the tight collaboration between car manufacturers and suppliers in the automotive industry, the customers (i.e., the car manufacturers) are very familiar with the implementations of a supplier and vice versa. A consequence of this is that change requests may be formulated on very different levels of abstraction. P7 says, *"Sometimes, the requirements are already an exact solution, sometimes they are just a rough idea on some desired functionality."*

Hierarchical structures *Hierarchical structures* were mentioned six times in the interviews as means to cope with complexity (P1, P5, P6, P7, P9). P6 explains, *"Especially, if there are long and complex relations between an input and an output signal, a hierarchical structure facilitates the calibration of a function."*

Structuring modes and phases Another issue that was mentioned in the interviews (P3, P4, P7, P8, P9) was *structuring modes and phases* of a function. Describing the lifecycle of a function in terms of abstract states was considered to have a positive impact on the understandability. P3 says, *"In most cases an abstract state-based view onto a function is sufficient to understand the general purpose of it."* Another aspect mentioned in the interviews was to define dependencies to other functions based on modes. P7 explains, *"A state-based representation of dependencies between functions is more suitable than a description solely based on data exchange."*

6.4.3 Relations between Challenges (RQ3)

We related the codes to each other and annotated the relations with explanations. We will present the most relevant results of this process by describing three stories that summarize a set of codes and their relations. In the stories, codes are written in bold letters. Each story contains a hypothesis (the heading of the story) that is backed up by the relation of codes within the story that follows the hypothesis.

Story 1: Comprehending a function is supported by documenting the basic principle of operation and its logical modes of operation To comprehend a function, a **high-level description of the function** is needed. This description should **highlight the basic principle of operation**. **A pure documentation of the code is not sufficient** as it hides the underlying physical principles. However, the basic principle of operation describes, in many cases, only the behavior of a function, when all enabling conditions are already fulfilled. **The logic that is embedded in the numerous enabling conditions is often neglected** in the function documentations. For the comprehension of a function, these logical conditions must be part of the function documentation. The description of these logical conditions can be supported by **structuring the functionality according to modes of operation**.

Story 2: Calibration of a function is supported by a clear description of the black-box interface of a function For the calibration of a function, the **function parameters have a strong influence on the behavior**. The logic and logical conditions of a function are also highly parameterized and the **logic constitutes a large part in many functions**. Logical conditions hamper the traceability of signal flow through a function design. However, **tracing signal flow is an important activity** for calibration engineers. **Understanding the black box interface** of a function is another way of tracing signals that are passed through a function. Therefore, the black box interface description of a function should contain the **most relevant signals for the physical principle of operation** as well as the signals influencing the logical conditions of the function.

Story 3: Logical conditions should be integrated into function documentations **The code of a function contains more details** than it is documented in the function documentation. In particular, logical conditions are often only contained in the code. These information about enabling conditions and logical interactions need to be integrated into the function documentation because **a pure documentation of the code is not sufficient**. **The logic constitutes for a large part in many functions** and thus it is even more necessary to integrate this part conveniently into function documentations.

6.5 Threats to Validity

A potential threat to the credibility of our findings is that we misinterpreted the data we collected and thus derived a theory that does not fit the data. To mitigate this threat, we performed *data triangulation* by reflecting our findings with two of existing function documentations and *member checking* by presenting and discussing our findings and interpretations with the interview participants. We conducted the study over a period of eight months, in which findings and interpretations were constantly discussed and applied to existing data. Another threat is a possible bias of the researcher. We tried to mitigate this threat by *making the researchers' intentions clear* and *co-opting* two additional researchers from the automotive company who participated in the interviews and the discussions of findings [Onwuegbuzie and Leech, 2007]. We used the frequency of code occurrences as an indicator for importance. This might be wrong since "*challenge x* is not important" might contribute to the importance of a code related to *challenge x*. We tried to mitigate this threat by separating negative quotations related to a code into another code (e.g., *code documentation is not sufficient*). The interviews and the coding and categorization process were conducted in German and translated only for the presentation in this thesis. To mitigate the threat of incorrect/improper translations by the researchers, the translations were checked by experts of the domain.

We conducted the interviews only in one company, which poses a threat to the generalizability of our findings. A replication study in other automotive companies has not yet been conducted. The relatively small group of only 9 interview participants might not be representative. Further participants may add new challenges and aspects. However, we are confident that our data base is saturated to a certain extent as we purposely selected participants with different roles and the ratio of codes that first appeared in one of the two last interviews was below 5%, i.e., not much additional data was added in the last interviews. We validated our findings by examining two exemplary function documentations from the engine control domain. While these, at least, cover two different types of functions (a feedback control function and a learning adaptation function), the general applicability of the conclusions to all types of functions cannot be assured.

6.6 Discussion and Implications

From the results of the study, we draw two conclusions:

Conclusion 1: A high-level description of the basic principle of operation of a function is necessary to comprehend a function. This high-level description should also incorporate the logical conditions that influence the function behavior (related codes: *Highlight the basic principle of operation*, *High-level description of a function*).

Conclusion 2: In the past, the complexity of a function was mainly driven by its control engineering part. Nowadays, the logical part of a function, i.e., its enabling conditions, its modes of operation, etc., accounts for an increasing part of the complexity. There are even functions that are trivial in the controlling of some values but highly complex in the situations, in which this controlling should be applied. This logical complexity should be considered in the function documentations (related code: *Structuring modes and phases*).

Especially the latter conclusion is well illustrated by an anecdotal example: In automobiles, there is a number of software functions, which, over time, learn an adaption factor and apply it to the measured value of a specific sensor to compensate for incorrect measurements for example due to dirt accumulation. Specifying this learning and application process is relatively easy in terms of control engineering theory (e.g., by giving a differential equation). However, for this function additional complexity is added, when it comes to specifying the conditions that need to be fulfilled before the learning process can be started. Furthermore, these conditions may also require interactions with other functions (e.g., requesting the permission of a scheduler). Before applying the actual control cycle, the function needs to pass several phases and conditions that account for a large part of the complexity. We speak of the logic of a function. In many function documentations, this logical part is just somehow added to the initial control engineering model, which does not really cater for such kinds of specifications.[1] The phases and states of the function are no longer comprehensible in the resulting function documentation.

6.7 Methodical Guidance for Function Documentations

The conclusions particularly address the way in which function documentations should be structured and which information they should contain. We extracted best practices and solutions for the mentioned challenges from the study results and developed a new artifact structure for function documentations. We applied this new function documentation structure to two exemplary functions from the engine control domain covering a feedback control function and a learning adaptation function. The resulting new function documentations were used to evaluate the findings and implications of our study (see the study execution process in Section 6.3.4).

The improved documentation structure introduces two new structuring means that were not part of the function documentation before: A function description on *three levels of abstraction* and *modes of operations* that structure the specification according to states or phases of a function. The structuring means and their integration into an improved documentation template are described in the following.

6.7.1 Three Levels of Abstraction

The idea of abstraction levels addresses the hypothesis of conclusion 1 by describing a function on different levels of abstraction to provide a function description without any technical or implementation details. Along these levels, high-level descriptions can be traced down to lower level requirements. Based on the interviews, we propose to specify requirements of a function according to the following three levels of abstraction.

[1] We have seen, for example, how logical conditions are integrated in dataflow models by a sequence of logical operator blocks like AND, OR, NOT.

Goals and High-Level Requirements: Describe the purpose of a function without any details of its realization. These requirements are the input for a function developer, who designs a general solution concept (basic principle of operation) based on these requirements.

Function Requirements: Requirements that are derived from the general solution concept the function developer creates. Function requirements are independent from the realization within the software architecture.

Software Implementation Requirements: Requirements that are derived from the function requirements. They describe the realization of the function requirements within the software architecture.

In current documentations, functions were only considered on one (very technical) level of abstraction, which lead to the above-mentioned problems (especially for comprehending a function).

6.7.2 Modes of Operation

The modes of operation address the hypothesis of conclusion 2. They reflect states or phases of the solution concept. These modes are a structural element that is used to structure the function requirements and the software implementation requirements. In addition to the modes, mode transitions are defined that specify the conditions that account for a mode change (cf. [Broy, 2010b; Dietrich and Atlee, 2013]). Structuring a function according to its phases or states is currently not part of the function documentations at Bosch (see Section 6.1). An interesting point for discussion is the relation between these modes of operation of a function and the modes in our artifact model (see Section 5.1.2). The modes of operation considered in this study primarily describe states or phases of one specific function, while we introduced the modes in our artifact model as modes characterizing a system. However, as also stated by Dietrich and Atlee [2013], function-specific modes may serve as an interface for describing function dependencies (e.g., "If the electronic brake function is *deactivated*, the cruise control function should also be *deactivated*"). In such cases, function-specific modes become relevant for specifying the behavior of the integrated system and, thus, should be added to the mode model of our RE methodology. In the study that we will present in the next chapter of the thesis, we elicited a mode model for a productive system and discovered that 37% of all elicited modes were function-specific.

6.7.3 Function Documentation Template

Figure 6.5 shows how we integrated the three levels of abstraction and the modes of operation into a function documentation template. We illustrate this integration by a simplified example of a sensor heating function, i.e., a function that, when active, heats a sensor device to a constant temperature and maintains this temperature on the desired level. The template presents the corresponding function documentation with 7 chapters. The content of Chapters 1 and 6–7 is similar to the content of current function documentations in automotive companies (see Section 6.1). They contain basic information such as authoring information (*Function Description*), the documentation of the actual realization (*Software Structure Documentation*), and a tab-

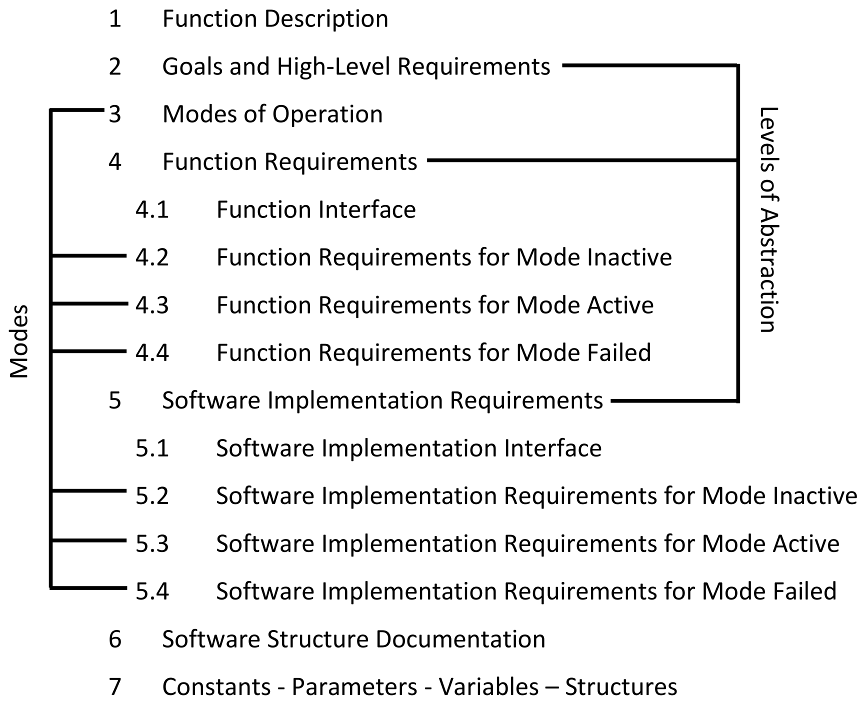

1 Function Description
2 Goals and High-Level Requirements
3 Modes of Operation
4 Function Requirements
4.1 Function Interface
4.2 Function Requirements for Mode Inactive
4.3 Function Requirements for Mode Active
4.4 Function Requirements for Mode Failed
5 Software Implementation Requirements
5.1 Software Implementation Interface
5.2 Software Implementation Requirements for Mode Inactive
5.3 Software Implementation Requirements for Mode Active
5.4 Software Implementation Requirements for Mode Failed
6 Software Structure Documentation
7 Constants - Parameters - Variables – Structures
Modes
Levels of Abstraction

Figure 6.5: Table of content of the function documentation template. Chapters 2, 4 and 5 represent the function on different levels of abstraction. Chapters 3, 4.2, 4.3, 4.4 and 5.2, 5.3, 5.4 structure the levels of abstraction by modes of operations.

ular representation of all signals used (*Parameters/Variables/Constants*). Chapters 2–5 are new and describe the requirements and interfaces of the function on the *three different levels of abstraction* and structured according to the *modes of operation*. In our example, the function has three modes of operation (*Inactive*, *Active*, and *Failed*). The three levels of abstraction each account for one chapter in the documentation template. The modes of operation structure each level in a similar way and thus relate the different levels of abstraction with each other.

We can classify the information provided in these chapters with respect to our semantic model introduced in Chapter 2. Figure 6.6 shows the relation between the chapters and the content they express. We see that the improved function documentation emphasizes the requirements of a function on different levels of abstraction explicitly. Compared to the initial documentation structure, as illustrated in Figure 6.1, we additionally document content of the mode model that describes the usage context of a function and structures the requirements. Via the mode model, it is also easier to comprehend the documented function as part of a function architecture because possible dependencies to other functions are made explicit by referencing the mode model in the requirements.

While the additional chapters may give the impression that the size of function documentations strongly increases, we observed only a moderate increase of 18% respectively 25% more pages in the improved documentations for two examples we considered in our study. Yet, these numbers are only first indicators and additional work is necessary to investigate the increase in size and effort, for example, with respect to the number of modes.

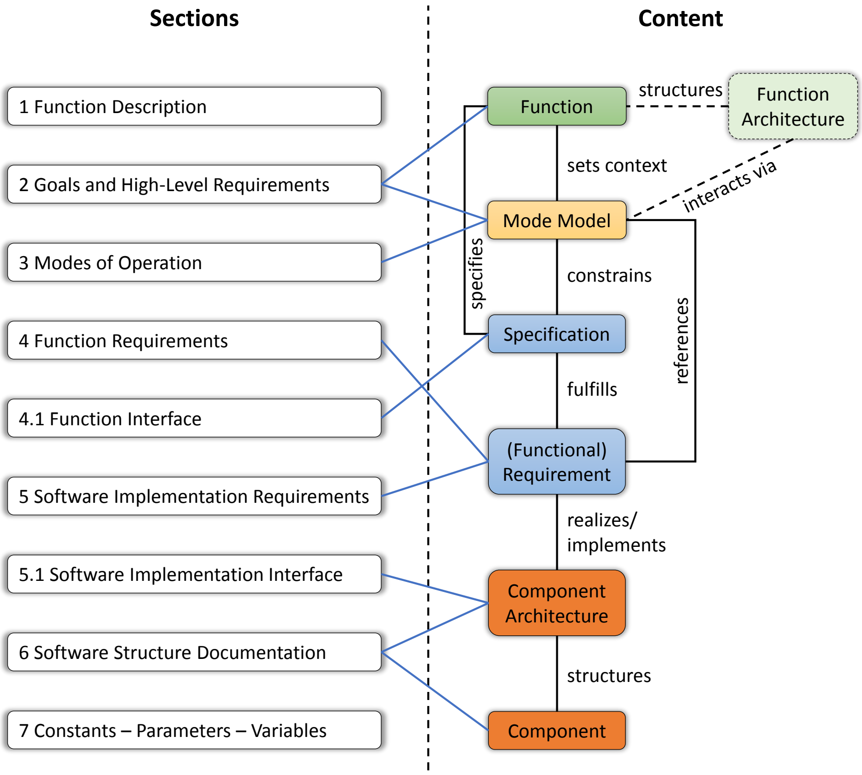


Figure 6.6: Sections of the improved function documentation template related to the content they describe.

ID:	2.1	**Title:**	Regulation of sensor temperature
Description:		The heating control of the sensor regulates the sensor temperature to a constant value.	
Rationale:		The sensors only work effectively when heated to a specific temperature.	

Figure 6.7: High-level requirement for a sensor heating function.

Please note, that the documentation structure is not restricted to be used in paper-based documentations. The introduced template could also be used as a schema for a model repository, a product lifecycle management (PLM) system, or as requirements structure in a requirements management system such as DOORS[2].

Goals and High-Level Requirements

Goals and high-level requirements capture the purpose of the function. These requirements are the input for a function developer. High-level requirements can have a *rationale* attached to it, which serves as an explanation for the requirement's origin. The requirements can be documented by a table as shown in Figure 6.7. The example shows the requirement's *id*, its *title*, and a *description* that covers the content of the requirement. The requirement states, as a high-level goal for a function, to regulate

[2] http://www.ibm.com/software/products/us/en/ratidoor/

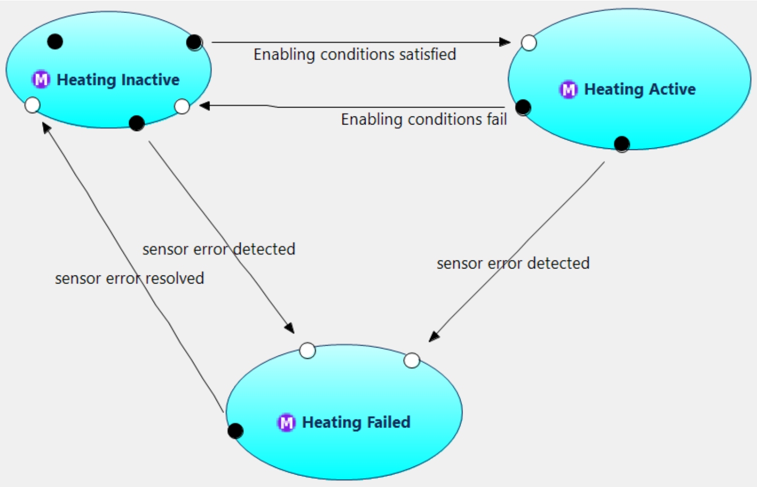


Figure 6.8: Modes of operation for a sensor heating function.

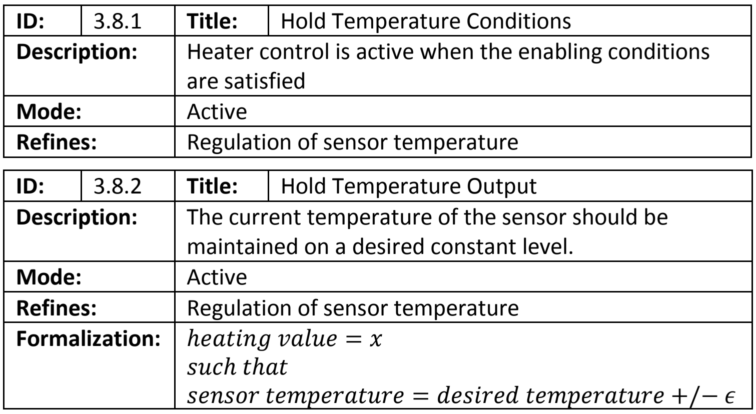
the temperature of a sensor to a constant value. The *rationale* justifies this goal by giving a reason for it. A function can have several goals, which might also contradict or influence each other (cf. [van Lamsweerde, 2001]). An alternative/additional description technique that can be used in this section is a *use case table*, as suggested in Section 5.1.1.

Modes of Operation

The modes of operation are determined by the general solution concept, which is created by the function developer. The modes of operation can be represented, for example, by a state machine that illustrates the modes and the transitions between them. Figure 6.8 shows the modes of operation for the sensor heating function. In the example, the function is structured into three modes, which represent different phases and situations of the function. In this example, we followed the approach to structure a function into three high-level states *Inactive*, *Active*, and *Failed* as proposed by Dietrich and Atlee [2013]. In real examples, this basic structure was too general in most cases. Almost all states had to be defined in more detail to provide an expressive model of the general solution concept. During the *Active* state, for example, a function might run through a number of additional states or phases. Additionally, a function might have a number of different degraded behaviors depending on the failure detected. However, in the examples we considered in our study, we were able to describe a function with no more than ten states. The function and software implementation requirements are structured according to these modes. The transitions between the modes of operation are described only informally in the figure. Details on the exact conditions for taking these transitions are specified in the function requirements.

Function Requirements

Function requirements document the general solution concept created by a function developer. They describe the solution concept independent from any hardware



ID:	3.8.1	**Title:**	Hold Temperature Conditions
Description:		Heater control is active when the enabling conditions are satisfied	
Mode:		Active	
Refines:		Regulation of sensor temperature	

ID:	3.8.2	**Title:**	Hold Temperature Output
Description:		The current temperature of the sensor should be maintained on a desired constant level.	
Mode:		Active	
Refines:		Regulation of sensor temperature	
Formalization:		$heating\ value = x$ $such\ that$ $sensor\ temperature = desired\ temperature +/- \epsilon$	

Figure 6.9: Function requirements for a sensor heating function.

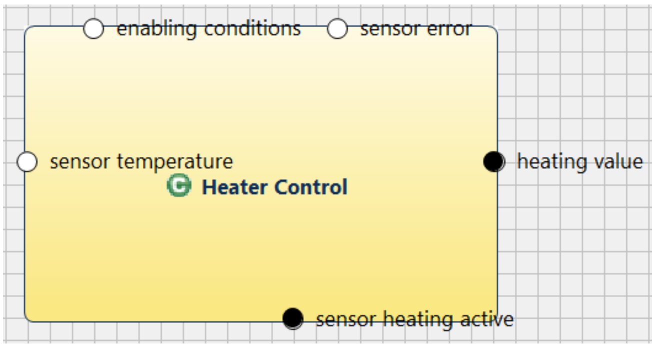
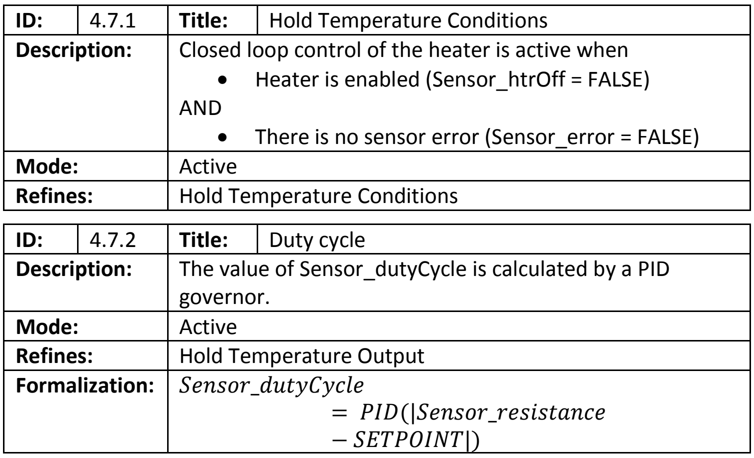


Figure 6.10: Function interface of the sensor heating function.

specifics and especially independent from the realization within the software architecture. Each function requirement refines a high-level requirement, which is indicated by a *Refines* attribute of the function requirement. Additionally, a *Formalization* attribute may contain a formalization, which relates inputs and outputs of the black-box interface of the function. In Section 5.1.3, we proposed *interface assertions* as a model type for formalizing such textual property descriptions. A function requirement may also have a *Mode* attribute, which indicates that this requirement is only valid when the function is in a certain mode of operation. Figure 6.9 gives two examples for function requirements of the sensor heating function. Both function requirements relate to the *Active* mode and refine the high-level requirement *Regulation of sensor temperature*.

Function Interface The function interface section is a subsection of the function requirements chapter and contains all inputs and outputs of the function that are referenced in the function requirements. Figure 6.10 shows the function interface for the sensor heating function. The main input and output quantities the function processes are denoted to the left and to the right of the function block. On top of the function block, the conditions that influence a mode transition of the function are denoted. Mode transitions of other functions that are affected by this function are denoted as outputs at the bottom of the function block. The inputs and outputs of the function interface can be used to formalize function requirements.



ID:	4.7.1	**Title:**	Hold Temperature Conditions
Description:		Closed loop control of the heater is active when • Heater is enabled (Sensor_htrOff = FALSE) AND • There is no sensor error (Sensor_error = FALSE)	
Mode:		Active	
Refines:		Hold Temperature Conditions	

ID:	4.7.2	**Title:**	Duty cycle
Description:		The value of Sensor_dutyCycle is calculated by a PID governor.	
Mode:		Active	
Refines:		Hold Temperature Output	
Formalization:		$Sensor\_dutyCycle = PID(\|Sensor\_resistance - SETPOINT\|)$	

Figure 6.11: Software implementation requirements for a sensor heating function.

Software Implementation Requirements

Software implementation requirements are derived from the function requirements. They describe the realization of the function requirements within the software architecture and under the condition of a specific hardware/software design, i.e., which specific hardware is used. Each software implementation requirement refines a function requirement, which is indicated by the *Refines* attribute of the software implementation requirement. Software implementation requirements are also structured by modes, indicated by the *Mode* attribute. Figure 6.11 gives two examples for software implementation requirements of the sensor heating function.

Software Implementation Interface Similar to the function interface, there is also a black-box interface description on the software implementation level. The software implementation interface contains inputs and outputs of the function on the level of the software architecture, i.e., the names and types of the main quantities and the logical signals that reflect the mode transitions as they are processed in the software architecture. Figure 6.10 shows the software implementation interface for the sensor heating function. As shown in the example, the signals of the function interface are mapped to signals of the software implementation interface. These mappings can be simple 1:1 mappings of abstract signals to concrete signals within the software architecture, or they can be more complex mappings. In the example shown here, the abstract output signal *heating value* of the function interface is mapped to the output value *SensU_dcycHt*, which controls the duty cycle of a heating device. This mapping encodes the decision/constraint that this specific heating device is controlled via a pulse-width modulation. If this sensor is replaced by another sensor with a different controlling mechanism, only the software implementation interface and the mapping has to be adjusted but the function interface and all function requirements are still valid.

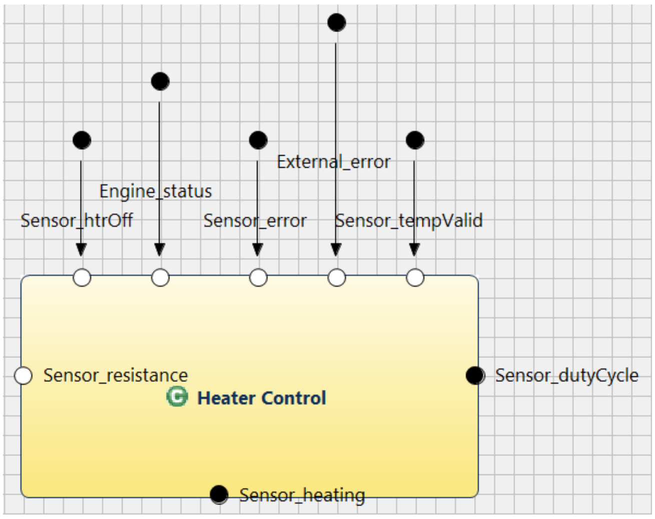


Figure 6.12: Software implementation interface of the sensor heating function.

6.8 Summary

In this chapter, we reported on an interview-based study we conducted to explore and expose challenges and shortcomings in the current documentation of automotive functions with the goal to derive methodical guidance and best practices for the documentation of functions. As a result of this study, we identified two major problems in function documentations, namely the separation of the basic principle of operation from the technical implementation and the inadequate specification of logical conditions, states, and phases of a function. To overcome these problems, we propose an improved documentation structure for functions and exemplify this structure by a function documentation template that describes a function on three levels of abstraction (goals, function requirements, and software implementation requirements) and structures its requirements according to modes of operation. This documentation structure provides guidance for a systematic documentation of functions, which are central artifacts of our RE methodology (see Section 5.1).

The central role of the concept of modes in the proposed documentation template and in our methodology in general poses the question, where these modes come from, how they are elicited and even if it is possible to specify them in a systematic manner. A method is needed to support the elicitation and specification of modes and we need to show that this method is feasible for realistic systems and delivers reasonable and manageable modes and mode models. Such a method and study will be described in the following chapter.

Chapter 7

Systematic Elicitation of Mode Models for Multifunctional Systems

The mode model has a central role in our RE methodology. In the presented case studies, we presented examples of modes and mode models. However, these modes were mainly defined in an *ad hoc* manner (e.g., by adding a mode when needed for modeling a function dependency). These procedures were sufficient for describing the function dependencies in the case studies; however, we think that a complete and consistent mode model has a positive impact also on the development of new functions and the evolution of existing ones. Systematic elicitation approaches are necessary to reach these goals. A second question that arises from the limited size of the presented case studies and examples of this thesis is how large a mode model for a realistic system can get. This has an impact on the scalability of our RE methodology. A third issue that has not been investigated in depth is the question what characterizes a mode and if there is an elicitation approach that is more likely to produce a mode model that fits our RE methodology.

This chapter presents three approaches for a systematic elicitation of mode models for a multifunctional system. The approaches are described in Section 7.1. We applied the three approaches in a case study to a productive automotive system to assess the approaches with respect to feasibility and differences between the resulting mode models. The results of this case study are presented and discussed in Section 7.2.

This chapter is partly based on a previous publication [Vogelsang et al., 2015].

7.1 Mode Models and Elicitation Approaches

The IEEE standard 29148 for software requirements specifications (SRS) notes: *"Some systems behave quite differently depending on the mode of operation. For example, a control system may have different sets of features depending on its mode: training, normal, or emergency."* [ISO/IEC/IEEE, 2011b]. The SRS proposes a specification structure for systems with modes in its annex. The use of statecharts [Harel, 1987] or other state-

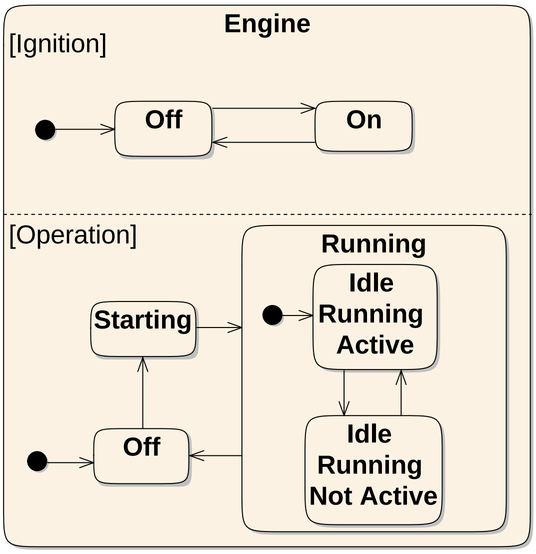


Figure 7.1: Mode chart representation of an exemplary mode model.

based specification techniques inherently relies on the advantages of structuring a system or a problem domain according to (observable) states. Use cases or scenarios may reference states or modes as trigger, pre- or postcondition (see Section 5.1.3).

We call the set of all modes that are used to formulate requirements the *mode model* of a system or a problem domain in general. Modes can be used explicitly (e.g., the mode *Engine Off* in a statechart) or implicitly (e.g., the term *engine is running* in the informal requirement "While the engine is running, the cooling control must maintain the motor temperature to a constant level").

While many specification techniques, including the RE methodology proposed in this thesis, rely on the notion of modes or states, there are no systematic approaches on how to elicit a set (a model) of modes/states for a system. Especially multifunctional systems may have a large number of states to consider.

As described in Sections 2.2.2 and 5.1.2, we represent a *mode* by a name and a set of *mode values* (e.g., $Operation = \{Off, Starting, Running\}$). These modes are summarized in a *mode list*. A *mode model* structures and formalizes the mode list. In a mode model, the modes of the mode list and their mode values are structured by defining the $children$ and $kind$ relation described in Section 2.2.2. Figure 7.1 shows an excerpt of an exemplary mode model. Note that in a mode model, we do not distinguish between modes and mode values anymore because by the hierarchical structure, a mode value may be a mode for itself with mode values (e.g., *Running* is a mode value of *Operation* but also a mode for itself with mode values *Idle Running active* and *Idle Running not active*). Instead, we call each element of a mode model a mode and distinguish them by their $kind$. Modes with $kind = AND$ are called *mode categories* because these modes just summarize a set of modes by a specific name/topic (e.g., *Engine* is a mode category in the example). Modes with $kind = BASIC$ correspond to mode values of the mode list, while modes with $kind = OR$ correspond to original modes of the mode list. These are the mode types defined by a mode model (see Section 2.2.2).

We aim at exploring different approaches for eliciting modes of a multifunctional system. In this chapter, we will investigate three approaches: (1) a deductive top-down approach, where modes are defined based on the domain knowledge of experts, (2) an inductive bottom-up approach, where modes are extracted from a given architecture of logical components and an analysis of dependencies between functions within this architecture, and (3) an analytical approach, where requirements are manually inspected for implicit or explicit references to modes.

7.1.1 Elicitation by Interviews with Domain Experts

We assume that domain experts have a good intuition what should be considered as a mode for a system based on their experience in the development of functions of a specific domain. Therefore, in this first elicitation approach, we obtain a mode model from interviewing domain experts. In the interviews, the interviewees are asked to enumerate what they consider as modes or states of the system under consideration. The result is a mode model that contains all modes mentioned in the interviews.

7.1.2 Elicitation by Function Dependency Analysis

Recent work of Broy [2010b] and Dietrich and Atlee [2013] postulate a relation between function dependencies and modes of a system. Therefore, this second approach is based on the results of a function dependency analysis. In Chapter 4, we presented the results of a feature dependency analysis performed on a productive automotive system. The analysis is based on a specific architecture of a system. In this architecture, logical components describe the realization/implementation of a function in a purely logical fashion, i.e., without any information about the hardware the system runs on. A network of logical components describes the steps that are necessary to transform the input data into the desired output data. An example for a system that consists of 3 functions (VF) that are realized by a network of 6 logical components is illustrated in Figure 7.2. The logical components are afterwards deployed to a set of electronic computing units that execute the behavior of the logical components.

In the dependency analysis, a function dependency is extracted from the function architecture if there is a logical component associated to one function exchanging data with a logical component associated to another function (e.g., communication between LC_3 and LC_4). From this dependency analysis, we obtain a list of dependencies between functions. Every item in this list contains two functions (source and target) and the data signal that is responsible for the dependency.

Now, for this study, we assume all of these dependencies to be candidates for modes: Each data signal represents a mode with the possible values of the data signal as mode values. A mode model is elicited by considering all dependency signals as modes and structuring them into mode categories.

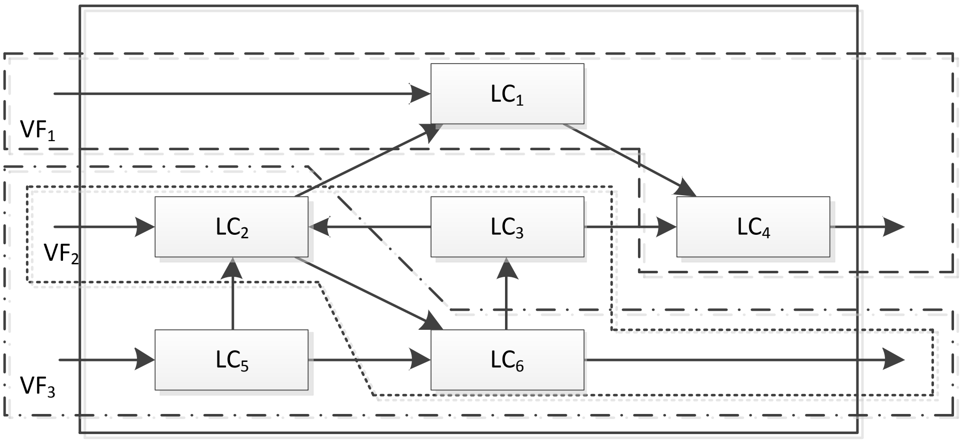


Figure 7.2: The logical components (rectangles) are connected by data channels (black arrows) and form a component architecture of the system (outer rectangle). Functions (VF) crosscut this architecture by the set of logical components that contribute to their realization (dashed forms).

7.1.3 Elicitation by Requirements Inspection

The third approach is based on natural language requirements, which are documented in requirements specifications. These specifications often implicitly contain statements about modes or states of a system. For example, the requirement "The air conditioning must maintain the desired temperature if the engine is running", refers to a mode of the engine, namely that the engine is in mode *running*. If a requirement refers to a mode value, we extracted this mode value in this elicitation approach. Afterwards, the extracted mode values are grouped into modes (e.g., mode values *running* and *off* are grouped to mode *engine state*). For the sake of clarity, the modes are finally structured into mode categories.

7.2 Case Study: A Mode Model for an Automotive System

7.2.1 Research Objective

Our study aims at exploring the different elicitation approaches for mode models of a multifunctional system and assessing the resulting models in terms of feasibility and differences to each other in the context of multifunctional systems.

7.2.2 Research Questions

RQ1: Are the introduced elicitation approaches feasible in practice?
We are interested whether the proposed elicitation approaches are capable of extract-

ing modes for multifunctional systems in practice. The results of this research question have an impact on the process of eliciting modes.

RQ2: Are the resulting mode models manageable?
It is an open question, how large a mode model can get for a realistic system. In this study, we want to assess the resulting mode models with respect to its size and manageability. The results of this research question have an impact on the scalability of mode-based specification approaches.

RQ3: How do the resulting mode models differ?
Different elicitation approaches may result in different modes and mode models. In this study, we want to assess the impact of the elicitation approach on the characteristics of the resulting modes. The results of this research question have an impact on the generalizability of a mode model and the selection of the appropriate elicitation approach.

7.2.3 Study Object

To answer the research questions, we applied the elicitation approaches in the context of the development of a truck at MAN Truck & Bus AG. Although the study objects for the three elicitation approaches were similar, they were not exactly the same. For the elicitation by interviews, we asked for modes relevant to the developer's daily work in general and not for a specific vehicle. For the elicitation by function dependency analysis, we analyzed a compact truck with a restricted set of 55 functions, while for the elicitation by requirements inspection, we examined requirements of a fully equipped heavy truck. This difference in the study objects was due to the availability of data at the time of the study execution. The threats that this poses to the validity of the results are discussed later.

7.2.4 Data Collection Procedures

As required for answering RQ1, we elicited a mode model for the analyzed vehicle systems through the three elicitation approaches described in Section 7.1. Each of the elicitation approaches results in a mode model that was afterwards analyzed to answer RQ2 and RQ3.

Elicitation by Interviews with Domain Experts

We conducted interviews with four developers of MAN Truck & Bus AG from the domains cabin & lights, base software, energy management, and driver assistance. These interview partners were selected by the head of the company's architecture group with the expectation to provide the maximum number of modes. Each interview lasted one hour. The stated modes were simultaneously written down by the interviewers and afterwards validated with the interviewees. The interviews were not structured by any questionnaire or guideline. However, we sometimes tried to guide the interviewee by asking open questions regarding specific classes of modes (e.g., "Are there any modes characterizing the vehicle's surroundings?"). After the four interviews, we documented the recorded modes in one integrated mode model containing all modes mentioned in the interviews.

Elicitation by Function Dependency Analysis

In Chapter 4, we already presented the results of a function dependency analysis for the study object at hand. From this dependency analysis, we extracted an overall of 91 different data signals responsible for all function dependencies. Each data signal is additionally characterized by a data type within the company's data backbone. Based on these data types, we derived a mode model from this list of data signals in the following way. For each signal, we checked the data type:

Boolean Signals: If the data signal had a Boolean data type, we defined a mode with the name of the signal and associated mode values *yes* and *no* (e.g., $BrakePedalPressed = \{yes, no\}$).

Enumeration Signals: If the data signal had an enumeration as data type, we defined a mode with the name of the signal and the enumeration members as associated mode values (e.g., $TransmissionMode = \{Park, Neutral, Drive, Reverse\}$).

Value Signals: If the data signal had a numeric data type such as km/h, we scanned the requirements specification of the corresponding vehicle function in order to find a *discretization* of the data signal. For example, in the requirement specification of one function, the signal *vehicle speed* was only used to distinguish between *low speed* and *high speed*. This is what we call a discretization. If we found a discretization of a numerical signal in the requirement specification, we defined a mode with the name of the signal and the extracted discrete values as associated mode values (e.g., $VehicleSpeed = \{low, high\}$).

If we were not able to transform a data signal to a mode by one of the three ways, we excluded this signal from the study. Similar to the proceeding for the modes of the interviews, we documented the resulting modes of the function dependency analysis in one mode model.

Elicitation by Requirements Inspection

For eliciting modes from their implicit usage within requirements, we inspected 223 requirements from a randomly picked sample of 11 vehicle functions. A requirement in our study is a textual statement consisting of 1–2 sentences in general. Similar to the proceeding for the modes of the other two elicitation approaches, we documented the resulting modes of the requirements inspection in one mode model.

7.2.5 Data Analysis Procedures

To answer the research questions RQ1 and RQ2, we analyzed the resulting mode models from the three elicitation approaches individually. For each mode model, we collected three measures: As a first measure, we counted the number of mode types, i.e., modes m with $kind(m) = OR$. The mode model shown in Figure 7.1, for example, has three mode types (*Ignition*, *Operation*, and *Running*). As a second measure, we determined, for each mode model, the number of mode values per mode type m, i.e., the number of elements in the set $\mathit{children}(m)$. In the example, the mode types *Ignition* and *Running* each have two mode values, while *Operation* has three. As a third measure, we determined the nesting depth of each mode type in a mode model. The nesting depth is determined by the length of the path through the hierarchical mode model to the mode type. In the example, mode type *Ignition* has nesting depth 2 (there is an additional *root* node in the mode model). These measures serve as indicators for the size and complexity of the resulting mode models. For both number of mode values and nesting depth, we show the distribution by a box plot diagram including median, maximum, and minimum values.

For the comparison of the resulting mode models (RQ3), we assess the three mode models with respect to equal mode types. We consider two mode types from different mode models to be equal if they have a similar name and at least one similar mode value. If, for example, in one mode model, there is a mode type $\mathit{Ignition} = \{\mathit{On}, \mathit{Off}\}$ and in another mode model there is a mode $\mathit{IgnitionState} = \{\mathit{On}, \mathit{Initializing}, \mathit{Off}\}$, we consider them as one mode type occurring in both mode models. The purpose behind this is that we are interested in classes of modes that are only elicited by a specific elicitation approach. We report on the number and ratio of modes elicited by just one elicitation approach and modes that were elicited by more than one elicitation approach (or even by all).

7.2.6 Validity Procedures

To ensure the validity of our results, we performed several validity procedures to mitigate mainly threats to the internal validity of the study.

The elicitation of a mode model by interviews poses the threat that the mentioned modes were incorrectly documented by the researchers. This could be due to imprecise or incorrect interpretation of what was being said or even by a bias of the researcher in any direction. To mitigate this threat, we took notes in the presence of the interviewees letting them intervene in case they found something was noted incorrectly or imprecise.

The extraction of a mode model based on a static analysis of function dependencies poses the threat that this automated analysis delivers incorrect or incomplete results that may then corrupt the resulting mode model. To mitigate this threat we selected the same study object that was subject to a former project that especially focused on this automated dependency analysis. In the context of this former project, the dependency analysis results were extensively reviewed and published [Vogelsang et al., 2012]. The extraction of the mode values for the dependency signals as described in Section 7.2.4 were taken from the company's database and are thus not subject to a researcher bias or interpretation.

The elicitation of a mode model by requirements inspection poses the threat that the extraction of modes is subject to the researcher's subjectivity. To mitigate this threat, we inspected a set of 50 requirements independently by two researchers classifying whether or not a requirement contains a mode. For this set, we observed an interrater agreement in terms of Cohen's kappa of 0.63 (*substantial agreement*) [Landis and Koch, 1977]. From this, we conclude that the extracted mode model is fairly reliable considering researcher's subjectivity. In addition, based on this double classification, we formulated a set of strict criteria that defined what is considered as mode in this study. These criteria were: A mode may change its mode value during runtime of the system (this excludes, for example, configurable parameters), a mode has a discrete set of mode values (this excludes continuous signals), a mode must be observable from outside the system (this excludes internal states), and a mode's granularity must maintain a specific notion of importance for the entire system (this excludes, for example, specific failure states that are only relevant to one specific function).

7.3 Threats to Validity

Despite the applied validity procedures, the study design poses some further threats to the validity of the results: The structure of the mode models, especially the summary of modes in mode categories was determined by the researchers and thus not strictly determined by the elicitation approach itself. However, this fact may only influence the measured nesting depth.

The low ratio of common modes for the elicited modes in the four interviews may suggest that additional interviews might lead to additional modes and thus the elicited mode model might not be complete. However, the interviewees were selected externally according to the head of the company's architecture department with the goal to cover a broad range of modes.

As described before, the study objects used for the elicitation approaches were not exactly the same. A consequence of this is that the absolute numbers of the elicited modes are not comparable between the different elicitation approaches. We will therefore not discuss differences of the elicitation approaches in terms of number of elicited modes.

For all of the three elicited mode models we cannot guarantee completeness of the elicited mode models because we only conducted four interviews, only inspected requirements from 11 functions, and only analyzed fully specified parts of the systems. Future work should investigate the completeness of the created models.

7.4 Study Results

In the first part of this section, we report on the elicited mode models for the three elicitation approaches (RQ1 and RQ2). In the second part, we will compare the resulting models and describe their similarities and differences (RQ3).

7.4.1 Elicited Mode Models

We were able to elicit reasonable mode models by all of the three elicitation approaches, i.e., none of the approaches was a dead end.

Table 7.1: Number of modes elicited in the interviews and ratio of modes mentioned in more than one interview (common modes).

Domain	Modes	Common Modes
cabin & lights	24	12 (50%)
base software	13	5 (38%)
driver assistance	11	3 (27%)
energy management	7	5 (71%)
Summary	**42**	**12 (28%)**

Mode Model from Interviews

Table 7.1 lists the number of elicited modes for each interview and the ratio of modes mentioned in at least one of the other interviews (*common modes*).

As shown in the table, the number of modes elicited from the interviews range from 7 to 24 and the ratio of modes that were also mentioned in another interview ranged from 27% to 71%. No single mode was mentioned in all of the interviews. The mode $Ignition = \{On, Off\}$ was the only mode that was mentioned in three of the four interviews. The final mode model that we assembled from all modes of the interviews contained 42 modes. 12/42 modes (28%) were mentioned in more than one interview. The 42 modes elicited from the interviews have 2 mode values on average (median) with a maximum of 6 mode values for one mode. The average (median) nesting depth of the modes in the resulting mode model is 2 with a maximum nesting depth of 4.

Mode Model from Function Dependency Analysis

We were able to transform 35/91 (38%) of the dependency signals from the analysis into an overall of 20 modes. The fact that there are less modes than transformed signals is due to signals that carried redundant information with respect to the mode (e.g., the signals *pedal pressed* and *pedal position* are both determined by the same mode *pedal state*). The dependency signals that could not be transformed into modes were numeric data signals for which we have not found any discretization in the requirements documents. The 20 extracted modes from the dependency analysis have 3 mode values on average (median) with a maximum of 15 mode values for one mode. The average (median) nesting depth of the modes in the resulting mode model is 2 with a maximum nesting depth of 3.

Mode Model from Requirements Inspection

In the inspection of requirements, we observed that 165/223 (74%) of the requirements contained information regarding at least one mode. From the analyzed requirements, we extracted 39 modes.

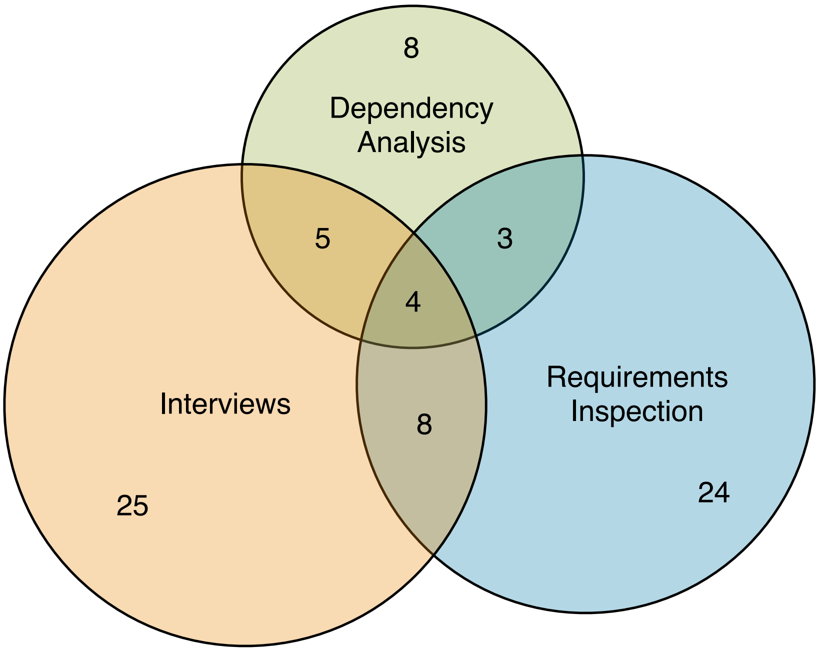


Figure 7.3: Proportionally correct distribution of modes with respect to the three elicitation approaches.

The extracted modes from the requirements inspection have 2 mode values on average (median) with a maximum of 7 mode values for one mode. The average (median) nesting depth of the modes in the resulting mode model is 2 with a maximum nesting depth of 3.

7.4.2 Comparison of Mode Models

The Venn diagram in Figure 7.3 shows the distribution of modes for the different elicitation approaches. The figure shows that 4 modes (5%) were elicited by all of the elicitation approaches. These modes are (1) the operation status of the engine, (2) the information whether the drive train is released, (3) the information whether the acceleration pedal is pressed, and (4) the current cruising range. An overall of 57 (74%) of all modes were elicited by only one elicitation approach.

The box plots in Figures 7.4 and 7.5 show a comparison of the number of mode values elicited for each mode and its nesting depth in the mode model of the different elicitation approaches.[1] The figure shows that the number of mode values for each mode is higher for the modes elicited by the dependency analysis, whereas for the other two approaches the number is similar. There is not much difference in the nesting depth of the models. There is only a slight tendency for the interview model that the nesting depth is 2 at minimum.

[1] The additional *Reference* plot can be ignored for now and will be discussed in Section 7.5

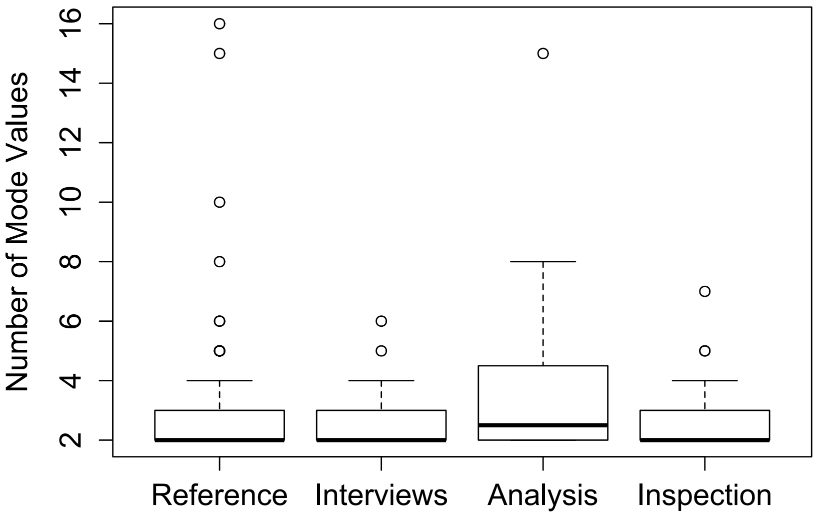


Figure 7.4: Box plot for distribution of mode values.

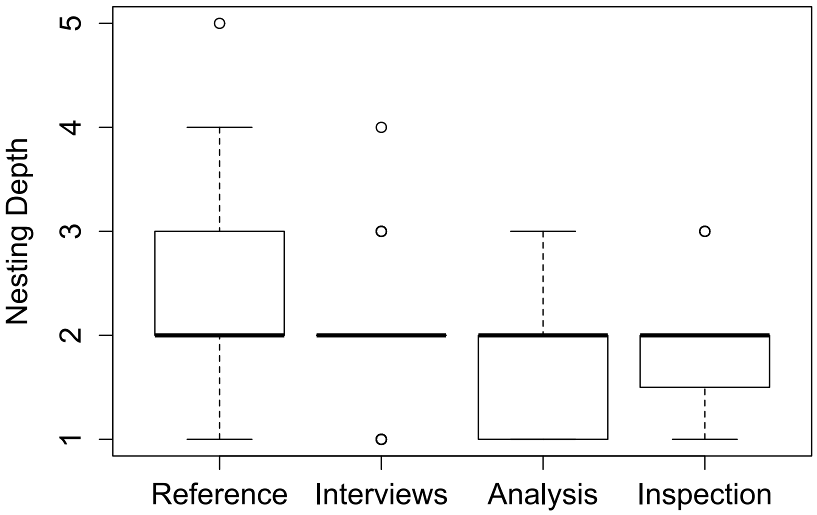


Figure 7.5: Box plot for distribution of nesting depth.

7.5 Discussion

In this section, we interpret the study results and discuss them with respect to the characteristics of the elicitation approaches, and the size of the mode models. Motivated by the diversity of the elicited mode models we also discuss a possible integration of the three mode models in a reference mode model.

7.5.1 Characteristics of the Elicitation Approaches

The goal of the study presented in this paper is to explore and assess different ways of eliciting a mode model for a multifunctional system. Based on the results, we discuss characteristics of the different approaches with respect to the type of modes that result from them. Therefore, we focus on the modes elicited by just one of the approaches and not by the others (see Figure 7.3).

The elicitation of modes based on **interviews** showed the highest number of modes elicited by just one approach (25). A reason for this may be that, in the interviews, we did not ask for a specific vehicle but for relevant modes in general. These modes were widely distributed over all kinds of mode categories. Interestingly, modes of the electronic infrastructure, such as the state of an electronic control unit or the activation of a bus system, were only elicited in the interviews but not in any of the other approaches. Many of the modes specific to the interview elicitation described rather abstract and general states, e.g., *bad weather condition* or *presence of oncoming vehicle*. Such modes seem to be important for the understanding of functionality by humans but may never appear as signals in the implementation. It might be interesting to manifest such abstract information also as explicit signals in the final implementation. The slightly higher nesting depth of the mode model elicited from the interviews compared with the other may indicate that it might be easier in this approach to determine structural relations between modes. This may be due to the fact that this approach is the only approach that exploits domain knowledge of experts, who can also share knowledge about structural relations between modes (e.g., one mode is a submode of another).

The mode model resulting from the **dependency analysis** is the smallest in terms of the number of elicited modes. One reason for this may be that we performed the dependency analysis on a compact truck with a restricted set of functions. Additionally, the number of elicited modes is influenced by the fact that, with this approach, only modes are elicited that originate from a dependency between functions of a system. Modes that are only relevant in one function cannot be elicited with this approach since they are not detected by the dependency analysis. The high number of dependency signals that could not be transferred into a mode (see Section 7.4.1) are an indicator that a large portion of function dependencies on an implementation/ architecture level are due to architectural or technical decisions and do not reflect dependencies on the level of functions, for which we assume that they correspond to modes. The dependency analysis, for example, also contained priority signals for the order of computation of functional blocks. This mismatch between dependencies on an implementation and function level is also addressed by the *optional feature problem* [Kästner et al., 2009]. The mode model resulting from the dependency analysis had the smallest number and portion of modes specific to that approach (8). What is striking for these modes is their detailed character. These modes described, for example, specific pressing scenarios for a pedal (e.g., *long-press* vs. *short-press*). Together with the fact that the number of mode values per mode in this approach is higher than in the other approaches, we conclude that this elicitation approach results in more detailed and more fine-grained modes.

The elicitation of modes based on **requirements inspection** resulted in 24 modes. Some of them described rather abstract and general conditions. This is similar to the modes of the elicitation by interviews and reflects that the requirements are written in the language of the domain experts. Another interesting aspect of the elicited modes is that they also cover modes specific to a certain function (e.g., *ABS active*). These function-oriented modes were not as extensively mentioned in the other approaches as in this approach.

In summary, we were surprised by the diversity of the modes resulting from the different elicitation approaches. Despite the quite different levels of abstraction, all modes have merit for describing a system's current state of operation.

7.5.2 Size of the Mode Models

A central aspect that we wanted to answer by this study is the question whether the elicitation of a mode model for an entire productive system results in a model consisting of hundreds or even thousands of modes that, in the end, is not understandable, usable, or maintainable anymore. By the results of our study, we are convinced that this is not the case. The mode models we elicited had 20 to 42 modes and even a combination of all mode models, which will be discussed in the next subsection, does not exceed 75 modes. An interesting question, which we have not addressed yet, would be to investigate the progression of the number of modes with respect to the number of functions.

An assessment of the elicitation approaches must also consider the necessary effort. Conducting interviews is time and person intensive; however, their analysis and the extraction of a mode model can be performed rather quickly. The extraction of modes from requirements is similarly time intensive but can be performed by a single (or a smaller group of) person(s). The extraction of modes by the dependency analysis can be performed automatically to a large extent.

7.5.3 Reference Mode Model

Motivated by the diversity of the elicited mode models (40–60% of the modes of one approach were specific to that approach), we created a *reference mode model* by merging the modes of all elicitation approaches. In the end, the reference model contained 75 modes that we structured into mode categories similar to our proceeding for the three elicitation approaches. Figures 7.4 and 7.5 show the distribution of mode values and nesting depth for the reference model. The nesting depth tends to be higher than in the original models. This is not surprising, assuming that a larger number of modes suggest additional mode categories to maintain the understandability. The number of mode values per mode is similar to those of the original models; however, there are a larger number of outliers with significantly more mode values than in

Table 7.2: Categorization of modes from the reference mode model with respect to their scope and if they relate to a function dependency.

		Dependency			
		yes	no	Σ	%
Scope	context	0	11	11	14.7
	system	8	28	36	48.0
	function	9	19	28	37.3
	Σ	17	58	75	
	%	22.7	77.3		100

the original models. We consider this reference model as a comprehensive model of operational modes for the development of a truck at MAN Truck & Bus AG since it considers and contains all viewpoints addressed by the three elicitation approaches.

7.5.4 Scopes of Modes

While assessing the elicited modes, we also noticed that modes can be classified with respect to the *scope* they address. The scope of a mode denotes the object of which a mode describes a property. To investigate and quantify this characterization in more detail, we classified the elicited modes of the reference mode model into the following three scopes:

context: Modes with this scope describe states of the operational environment of the system under consideration (e.g., $\mathit{AmbientTemperature} = \{\mathit{low}, \mathit{high}\}$).

system: Modes with this scope describe states of the system under consideration (e.g., $\mathit{DrivingDirection} = \{\mathit{forward}, \mathit{backward}\}$).

function: Modes with this scope describe states of specific functions of the system under consideration (e.g., $\mathit{CruiseControl} = \{\mathit{Off}, \mathit{Standby}, \mathit{Active}\}$).

With this classification, we can assess the quantitative relationship between function-specific modes, as discussed in Chapter 6 as part of the function specification, and system-wide modes, as originally introduced in our artifact model (see Section 5.1). More fine-grained or other classifications of mode scopes are also possible (e.g., scope with respect a domain of a car).

We performed the classification by two researchers. A first round of independent classification resulted in an inter rater agreement in terms of Cohen's Kappa of 0.54 (*moderate agreement*) [Landis and Koch, 1977]. Deviations in the classification only appeared between the *system* and the *function* scope. We resolved these deviations in a discussion. Table 7.2 shows the resulting distribution of modes of the reference mode model with respect to their scopes. Additionally, we correlated this distribution with the property whether the mode describes a function dependency. We assigned this property to the modes that originated from the dependency analysis. As discussed in the threats to validity, the absolute numbers may not be comparable but we can discuss the distribution with respect to the scope.

The table shows that system-specific modes account for the largest part of modes in the reference mode model. A slightly smaller number of modes have a function-wide scope. In our study, we elicited most of these modes by the requirements inspection approach. Only a small, but still considerable, number of modes have a context scope. When considering the function dependencies, none of the context modes relates to a function dependency. However, as we discussed before, this may result from the fact that these modes are not (yet) captured by signals within the implementation and therefore cannot be detected by our dependency analysis. The dependencies that we found in our analysis were almost equally distributed over modes with a system or function scope. This, additionally, provides a strong motivation for explicitly specifying states or modes of a function in their specification as we proposed in Chapter 6. These results coincide with the results of Dietrich and Atlee [2013] who also assessed a correlation between function-specific modes and behavioral function dependencies.

7.6 Summary

In this chapter, we presented three approaches for eliciting a mode model for a multifunctional system. The purpose of a mode model is to describe a system in terms of states. Such a mode model may serve as a basis for the proposed specification approach presented in this thesis or any other state-based specification of system functionality.

We applied the three elicitation approaches in the context of the development of a truck at MAN Truck & Bus AG. By application in this context, we answered our research questions in the following way:

RQ1: Are the introduced elicitation approaches feasible in practice?
Yes, in our study we were able to elicit modes through all three elicitation approaches. All approaches resulted in a mode model and are thus feasible.

RQ2: Are the resulting mode models manageable?
Yes, the size of the elicited mode models ranged from 20 to 42 modes and even a combination of all mode models (*reference mode model*) did not exceed 75 modes.

RQ3: How do the resulting mode models differ?
With all three approaches, we elicited modes that were solely elicited by one approach. We thereby conclude that none of the approaches was superfluous (e.g., if most of its modes were also elicited by another approach).

The aim of this study was to explore different elicitation approaches for mode models and assessing the resulting models. The RE methodology presented in this thesis shows how the resulting mode models can be integrated in a requirements engineering process. We see this as a major contribution to the modeling and formalization of requirements. Filipovikj et al. [2014], for example, mention the use of high-level concepts, such as *shutdown* or *start-up*, in textual requirements as an impediment to their formalization because these concepts are too abstract and ambiguous. A mode model may contribute to this formalization by providing high-level concepts with a precise meaning.

The results of this study provide further empirical evidence that explicitly specifying the modes of a function, as suggested by our function specification template described in Chapter 6, supports the specification of function dependencies.

In Section 2.2.2, we introduced mode models including mode transitions, which have not been tackled so far in our study. Adding mode transitions increases the expressiveness of a mode model but requires additional information that is not elicited by our approaches. More general, it might also be interesting to state invariants over the mode model (e.g., a specific mode combination must never occur). Starting the requirements engineering activities with a specification of a mode model for an entire multifunctional system may also lead to a completely new way of describing, specifying, and developing functionality.

Chapter 8

Conclusions and Outlook

In this final chapter, we summarize the contributions of this thesis and describe possible directions for further research.

8.1 Conclusions

This thesis is based on the problem statement *"We need a comprehensive requirements engineering methodology for multifunctional systems that integrates and supports the specification of function dependencies"* (see Section 1.2). We claimed that this thesis provides supporting evidence and solutions for the stated problem. In the following, we conclude that these claims are supported by the contributions provided in this thesis.

8.1.1 Significance of Function Dependencies

From the results presented in Chapter 4, we can answer the stated research questions:

RQ1: To what extent do dependencies between functions exist? Function dependencies are numerous and pervade the whole system. Our analysis of the function architecture of two productive automotive systems in industry shows that at least 69% of the analyzed vehicle functions depend on other vehicle functions or influence other vehicle functions.

RQ2: How are dependencies distributed over all functions? Function dependencies are distributed over the whole system. However, some functions are more central than others are. In our study, single vehicle functions had dependencies to up to 56% of all vehicle functions.

RQ3: What categories of data characterize the dependencies? The information that represents a function dependency is, in most cases, a value from a discrete finite set of possible fixed values. In our analysis, around 74% of the channels that represented a dependency had an enumeration or Boolean data type.

RQ4: To what extent are developers aware of function dependencies? In our study, we have seen that modeling the dependencies on an architectural level is insufficient for analyzing them, leading to a 50% chance that a developer was not aware of a specific dependency in the analyzed system. In our analysis, this was particularly striking when the function dependencies arose from architectural decisions.

Conclusion

The results of our study presented in Chapter 4 show that function dependencies pose a great challenge for the development of multifunctional systems. The extent and distribution of them are not reflected by the awareness of the developers. This explains why function dependencies are a major source of unintended and erroneous behavior and proves their significance for the specification of multifunctional systems. Considering these conclusions, it becomes obvious that it is necessary to integrate the modeling of function dependencies on the level of functions into a comprehensive specification methodology for such systems.

8.1.2 Requirements Engineering for Multifunctional Systems

Our empirical results show that function dependencies are numerous in productive multifunctional systems but are not considered in their specification. In Chapters 5 to 7, we presented an integration of a formal specification technique for multifunctional systems into a comprehensive requirements engineering methodology.

Integrating Functions and Modes into a Model-based RE Methodology

As a first step, we described an artifact model in Chapter 5 that introduces the artifacts and modeling concepts used in our methodology followed by a description of the role of these artifacts in a development process.

We presented two case studies, where the artifact model was instantiated and applied in a scenario-driven requirements engineering context and in a property-driven requirements engineering context. The case studies showed that the methodology is feasible and allows detecting inconsistencies in requirements early, especially with respect to function dependencies.

However, the case study evaluation also revealed the need for additional methodical guidance especially with respect to the level of granularity on which the artifact model should be applied and the decomposition and documentation of functions. Additionally, methodical guidance is needed for a systematic elicitation of mode models.

Function Documentations for Multifunctional Systems

Based on the results of the study presented in Chapter 6, we suggest documenting functions by applying the artifact model of our methodology on three different levels of abstraction: *Goals and high-level requirements*, which describe the purpose of a function without any details of its realization, *function requirements*, which are independent from the realization within the actual software architecture, and *software implementation requirements*, which describe the realization of the function requirements within the software architecture. The requirements within these levels of abstraction are structured with respect to *modes of operation*, which break a function down to a set of states or phases. These suggestions for documenting functions in a multifunctional system are derived from an exploratory qualitative study that revealed (1) a lack of separation of the basic principle of operation from the technical implementation of a function, and (2) an inadequate specification of logical conditions, states, and phases of a function in current function documentations.

Elicitation of Mode Models

We presented three approaches for eliciting a mode model for a multifunctional system. Mode models play an important role in the methodology of this thesis. The results presented in Chapter 7 show that we were able to elicit modes for an existing automotive system by means of all elicitation approaches. The resulting mode models contained 20 to 42 modes and even a combination of all mode models (reference mode model) did not exceed 75 modes. From these results, we conclude that it is possible to elicit manageable mode models for an entire system. Moreover, the results of the study provide further empirical evidence that explicitly specifying the modes of a function, as suggested by our function documentation template described in Chapter 6, supports the specification of function dependencies.

Conclusion

With the presented RE methodology and its evaluation in four case studies, we have shown that an integration of a formal specification technique into a comprehensive RE methodology is necessary to handle the challenge of specifying multifunctional systems. In the center of this methodology is the specification of functions, their structuring in a function architecture, and the explicit specification of function dependencies by means of a mode model. The contributions of this thesis provide an artifact model and analysis procedures that operationalize the formal specification approach presented by Broy [2010b]. Furthermore, we provide methodical guidance in the application of the artifact model for documenting functions of a multifunctional system. Lastly, we provide empirical evidence for the feasibility and usefulness of explicitly specifying a mode model for a multifunctional system and assess three elicitation approaches for such a mode model.

8.2 Outlook

This section discusses possible improvements of the presented work and illustrates directions for further research.

8.2.1 Nonfunctional Requirements

While the contributions of this thesis only consider functional requirements, an interesting research direction would be to extend the approach to nonfunctional properties of a system, which may influence the behavior of a system.

The basic assumption behind this idea is that so-called *nonfunctional* requirements, in fact, are always reflected in any kind of system behavior. This misconception of the term *nonfunctional* requirements is, for example, also criticized and discussed by Glinz [2007]. The approach of this thesis may serve as a blueprint for refining and formalizing also nonfunctional requirements by interpreting them as behavioral properties. High-level nonfunctional goals as they are stated, for example, in the ISO/IEC 9126 [ISO/IEC, 2001] (e.g., usability, reliability, efficiency), are refined to observations that can be defined in terms of observable system behavior. For some of these properties an extension of the used system modeling theory is necessary. Properties concerning the availability of a system are often defined by the use of statistical measures (e.g., the system must be available 99.9% of the time). Specifying such properties needs an extension of the modeling theory as proposed, for example, by Neubeck [2012]. How such an extended theory can be used to formalize availability properties is described by Junker and Neubeck [2012].

We think it is promising to examine also other nonfunctional properties and to assess whether and how they can be specified by means of our formal modeling theory. This way, the approach described in this thesis would also capture the specification of nonfunctional properties. An interesting, and not yet extensively discussed research question is: What is the relation between nonfunctional properties, functions, and function dependencies? It sounds reasonable to specify a nonfunctional property with respect to a function (e.g., "The *airbag function* should be available 99.9% of the time"); but how is this availability affected by the availability of other functions to which the airbag function has dependencies? In the work of Siegmund et al. [2012], the authors present an approach to predict program performance based on selected features. What is special about this approach is that it also takes the impact of feature interactions into account. The performance of a system is thus determined by performance measures of the single features and those of their interaction. In their work, the authors also provide a feature interaction detection approach that is based on performance measurements, i.e., nonfunctional behavior might be used to detect interactions between features. It is an interesting question how such an approach relates to the mode model that is introduced in this thesis. For example, is it possible to associate a mode with a performance measurement? Can mode observations be used to detect function dependencies? Such considerations may lead to a notion of nonfunctional feature interactions.

8.2.2 Natural Language Requirements

The approaches presented in this thesis propose to use appropriate models to capture and specify requirements. However, in practice, a large portion of requirements and system descriptions are captured as natural language text [Luisa et al., 2004]. In fact, we have seen in the studies that have been conducted as part of this thesis that a lot of important information is documented implicitly in natural language artifacts. This bears the risk that ambiguities or incomplete requirements specifications remain undiscovered because natural language has no formal semantics, and thus these issues are hard to detect.

An interesting topic for future research is the question how the proposed models of this thesis can be related to natural language representations. In the research community, there are different approaches that may contribute to this question. Natural language patterns, as for example proposed by Denger et al. [2003], restrict the use of natural language to a subset that allows a transformation to (semi)formal models. In Chapter 6, we introduced a documentation template for functions. Future research may investigate the use of natural language patterns as part of this function documentation template.

Especially in the presence of legacy systems and its documentation (if existent), it is also interesting to consider information retrieval techniques to (automatically) extract implicit knowledge from natural language artifacts and to transform it into models proposed in this thesis. In the context of the study presented in Chapter 7, we saw that natural language requirements were a fertile source of eliciting modes of a system. Kof and Penzenstadler [2011] propose a semi-automated approach to detect states in natural language requirements. It would be interesting to investigate this approach to extract system modes from natural language requirements.

8.2.3 Variability and Product Families

The focus of this thesis is on the specification of a single system. However, multifunctional systems are often part of a whole family of products that consists of a set of similar products. These products of a product family may differ, for example, in the set of functions they offer. A good example for this is a product line of an automotive company that contains a set of varying vehicle models (e.g., for different markets and in different luxury classes).

The close relation between function or feature modeling and modeling of variability is reflected by many approaches from the field of feature modeling (e.g., [Kang et al., 1990; Pohl et al., 2005; Shaker et al., 2012]). The whole field of feature-oriented software development considers features as the primary units of reuse, and the variants of a software system vary in the features they provide [Apel et al., 2013a].

The approach presented in this thesis may contribute to and extend these approaches by considering the variability of features/functions in a more differentiated way. Our approach, for example, allows differentiating variability in the set of functions, variability in the behavior of functions, variability in the influence of functions to each other, and, maybe most interestingly, variability in the set of modes. Variability of modes may be worth investigating in more detail in the future. For example, an additional safety mode that is required when operating a system in a specific country may have an influence on all functions of the system. For modeling this variability of modes in a mode model it might be interesting to investigate variability extension mechanisms for statecharts (e.g., as *parameterized* statechart models [Gomaa, 2005]). Shaker et al. [2012], for example, already use this technique in their feature-oriented requirements modeling language (FORML) to annotate statechart elements, such as states, labels, or transitions, with feature dependent parameters.

8.2.4 Software Product Management

Software product management of multifunctional systems is a challenging issue not least because of the need to handle short iterative lifecycles and families of these systems [Ebert and Brinkkemper, 2014]. After collecting experience in control engineering from a three-year collaboration in the automotive domain, we conclude that software product management is only weakly aligned with feature and platform development. This claim is also supported, for example, by a study of Lucassen et al. [2014], who report that software program managers and software architects collaborate in requirements gathering and refinement but do not use any structured models for this purpose. This situation makes it difficult to assure feasibility of innovations and to identify potentials for innovations.

For innovation management, which is part of product management, we already proposed a first step towards the integration of a model-based RE methodology as introduced in this thesis into the activities of innovation and technology management [Gleirscher et al., 2014]. We present how function architectures can be extended to represent also functional evolution over time. Technology roadmaps can be derived from this model. A central idea of this approach is to separate functional evolution from technological evolution. This separation is described by three models: the function architecture, which describes the evolution of functions offered to the customer, the platform service architecture, which describes the capabilities of the technology platform, and the platform component model, which describes the constituents of the technology platform. In our work, we illustrate how these models support two innovation scenarios (requirements-based and technology-based innovation). While we consider this approach promising for the connection of model-based development to earlier development phases from product and innovation management, the approach still needs to be refined and evaluated in more detail. Especially, the relation between evolution (of functions or platform capabilities) and variation is something that is not properly captured in current working practices.

Tran and Massacci [2014] present an approach for reasoning about feature model evolution. In their approach, they present two kinds of models: an *evolution possibility model* that describes possibilities for a feature model to evolve, and an evolutionary feature model that describes the feature model with all changes due to incorporated evolutions. Based on these models, they present, besides others, a *survivability analysis* that assesses whether a configuration (i.e., a set of features) could survive during the expected evolution. However, in their models and analyses, behavioral interactions between features are not considered. We are convinced that feature interactions are as important for the introduction of innovative functionality as the evolution of the isolated features. A large portion of innovative functionality comes from a smart interplay of existing features.

Bibliography

Acosta, R., Burns, C., Rzepka, W., and Sidoran, J. (1994). A case study of applying rapid prototyping techniques in the requirements engineering environment. In *Proceedings of the 1st International Conference on Requirements Engineering (RE'94)*.

Adam, S., Doerr, J., and Eisenbarth, M. (2009). Lessons learned from best practice-oriented process improvement in requirements engineering: A glance into current industrial re application. In *Proceedings of the 4th International Workshop on Requirements Engineering Education and Training (REET'09)*.

Adolph, S., Hall, W., and Kruchten, P. (2011). Using grounded theory to study the experience of software development. *Empirical Software Engineering*, 16.

Apel, S., Batory, D. S., Kästner, C., and Saake, G. (2013a). *Feature-Oriented Software Product Lines – Concepts and Implementation*. Springer.

Apel, S. and Kästner, C. (2009). Overview of feature-oriented software development. *Journal of Object Technology (JOT)*.

Apel, S., Kästner, C., and Batory, D. (2008). Program refactoring using functional aspects. In *Proceedings of the 7th International Conference on Generative Programming and Component Engineering (GPCE'08)*.

Apel, S., Kolesnikov, S., Siegmund, N., Kästner, C., and Garvin, B. (2013b). Exploring feature interactions in the wild: The new feature-interaction challenge. In *Proceedings of the 5th International Workshop on Feature-Oriented Software Development (FOSD'13)*.

Apel, S. and Lengauer, C. (2008). Superimposition: A language-independent approach to software composition. In *Software Composition*. Springer Berlin Heidelberg.

Apel, S., Lengauer, C., Möller, B., and Kästner, C. (2010a). An algebraic foundation for automatic feature-based program synthesis. *Science of Computer Programming*, 75(11).

Apel, S., Scholz, W., Lengauer, C., and Kästner, C. (2010b). Detecting dependences and interactions in feature-oriented design. In *Proceedings of the 21st International Symposium on Software Reliability Engineering (ISSRE'10)*.

Batory, D., Sarvela, J., and Rauschmayer, A. (2004). Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30(6).

Benz, S. (2010). *Generating Tests for Feature Interaction*. PhD thesis, Technische Universität München.

Böhm, W., Junker, M., Vogelsang, A., Teufl, S., Pinger, R., and Rahn, K. (2014). A formal systems engineering approach in practice: An experience report. In *Proceedings of the 1st International Workshop on Software Engineering Research and Industrial Practices (SER&IPs'14)*.

Böhm, W. and Vogelsang, A. (2013). An artifact-oriented framework for the seamless development of embedded systems. In *Software Engineering 2013 – Workshopband*, volume 215 of *Lecture Notes in Computer Science*. GI.

Booch, G. and Rumbaugh, J. (1997). *Unified Method for Object-Oriented Development*. Rational Software Corporation, version 1.0 edition.

Bouma, L. and Velthuijsen, H. (1994). *Feature interactions in telecommunications systems*. IOS press.

Brin, S. and Page, L. (1998). The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30.

Broy, M. (2005). A semantic and methodological essence of message sequence charts. *Science of Computer Programming*, 54(2–3).

Broy, M. (2006). Challenges in automotive software engineering. In *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*.

Broy, M. (2007). Model-driven architecture-centric engineering of (embedded) software intensive systems: Modeling theories and architectural milestones. *Innovations in Systems and Software Engineering*, 3(1).

Broy, M. (2010a). A logical basis for component-oriented software and systems engineering. *The Computer Journal*, 53(10).

Broy, M. (2010b). Multifunctional software systems: Structured modeling and specification of functional requirements. *Science of Computer Programming*, 75(12).

Broy, M., Damm, W., Henkler, S., Pohl, K., Vogelsang, A., and Weyer, T. (2012). Introduction to the SPES modeling framework. In *Model-Based Engineering of Embedded Systems*. Springer Berlin Heidelberg.

Broy, M., Krüger, I., Pretschner, A., and Salzmann, C. (2007a). Engineering automotive software. *Proceedings of the IEEE*, 95(2).

Broy, M., Krüger, I. H., and Meisinger, M. (2007b). A formal model of services. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16.

Broy, M. and Stølen, K. (2001). *Specification and development of interactive systems: Focus on streams, interfaces, and refinement*. Springer.

Calder, M., Kolberg, M., Magill, E. H., and Reiff-Marganiec, S. (2003). Feature interaction: A critical review and considered forecast. *Computer Networks*, 41(1).

Calder, M. and Magill, E. H., editors (2000). *Feature Interactions in Telecommunications and Software Systems VI*. IOS Press.

Cataldo, M. and Herbsleb, J. D. (2011). Factors leading to integration failures in global feature-oriented development: An empirical analysis. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*.

Chen, K., Zhang, W., Zhao, H., and Mei, H. (2005). An approach to constructing feature models based on requirements clustering. In *Proceedings of the 13th IEEE International Conference on Requirements Engineering (RE'05)*.

Clarke, E., Fehnker, A., Jha, S. K., and Veith, H. (2005). Temporal logic model checking. In *Handbook of Networked and Embedded Control Systems*, Control Engineering. Birkhäuser Boston.

Classen, A., Heymans, P., and Schobbens, P.-Y. (2008). What's in a feature: A requirements engineering perspective. In *Fundamental Approaches to Software Engineering*, volume 4961 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg.

Cockburn, A. (2001). *Writing Effective Use Cases*. Addison-Wesley Professional.

Corbin, J. and Strauss, A. (2008). *Basics of qualitative research: Techniques and procedures for developing grounded theory*. Sage.

Damm, W., Votintseva, A., Metzner, A., Josko, B., Peikenkamp, T., and Böde, E. (2005). Boosting re-use of embedded automotive applications through rich components. *Proceedings of Foundations of Interface Technologies*, 2005.

Davis, A. M. (1982). Rapid prototyping using executable requirements specifications. *SIGSOFT Software Engineering Notes*, 7(5).

DeMarco, T. (1979). *Structured analysis and system specification*. Yourdon Press.

Denger, C., Berry, D., and Kamsties, E. (2003). Higher quality requirements specifications through natural language patterns. In *Proceedings of the IEEE International Conference on Software: Science, Technology and Engineering (SwSTE '03)*.

Dietrich, D. and Atlee, J. M. (2013). A mode-based pattern for feature requirements, and a generic feature interface. In *Proceedings of the 21st IEEE International Requirements Engineering Conference (RE'13)*.

Dwyer, M. B., Avrunin, G. S., and Corbett, J. C. (1999). Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*.

Ebert, C. and Brinkkemper, S. (2014). Software product management – An industry evaluation. *Journal of Systems and Software*, 95(0).

Eshuis, R. (2009). Reconciling statechart semantics. *Science of Computer Programming*, 74(3).

Ferrari, A., Gnesi, S., and Tolomei, G. (2013). Using clustering to improve the structure of natural language requirements documents. In *Requirements Engineering: Foundation for Software Quality*, volume 7830 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg.

Filipovikj, P., Nyberg, M., and Rodriguez-Navas, G. (2014). Reassessing the pattern-based approach for formalizing requirements in the automotive domain. In *Proceedings of the 22nd IEEE International Requirements Engineering Conference (RE'14)*.

Fowler, M. and Scott, K. (2000). *UML distilled – A brief guide to the Standard Object Modeling Language*. Addison-Wesley-Longman, 2nd edition.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design patterns: Elements of reusable object-oriented software*. Pearson Education.

Gleirscher, M., Vogelsang, A., and Fuhrmann, S. (2014). A model-based approach to innovation management of automotive control systems. In *Proceedings of the 8th International Workshop on Software Product Management (IWSPM'14)*.

Glinz, M. (2007). On non-functional requirements. In *Proceedings of the 15th IEEE International Requirements Engineering Conference (RE'07)*.

Gomaa, H. (2005). Designing software product lines with UML. In *Proceedings of the 35th Annual IEEE Software Engineering Workshop (SEW'05)*.

Greiler, M., van Deursen, A., and Storey, M. (2012). Test confessions: A study of testing practices for plug-in systems. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*.

Greiler, M., van Deursen, A., and Storey, M.-A. (2011). What eclipsers think and do about testing: A grounded theory. Technical Report SERG-2011-010, Delft University of Technology.

Griss, M., Favaro, J., and d'Alessandro, M. (1998). Integrating feature modeling with the RSEB. In *Proceedings of the 5th International Conference on Software Reuse (ICSR'98)*.

Gross, A. and Doerr, J. (2012). What you need is what you get!: The vision of view-based requirements specifications. In *Proceedings of the 20th IEEE Requirements Engineering Conference (RE'12)*.

Gunter, C. A., Gunter, E. L., Jackson, M., and Zave, P. (2000). A reference model for requirements and specifications. *IEEE Software*, 17(3).

Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3).

Heitmeyer, C. L. (1998). Using the SCR* toolset to specify software requirements. In *Proceedings of the 2nd IEEE Workshop on Industrial-Strength Formal Specification Techniques (WIFT'98)*.

Heitmeyer, C. L., Archer, M., Bharadwaj, R., and Jeffords, R. D. (2005). Tools for constructing requirements specifications: The SCR toolset at the age of ten. *International Journal of Computer Systems Science & Engineering*, 20(1).

Heitmeyer, C. L., Kirby, J., and Labaw, B. G. (1997). The SCR method for formally specifying, verifying, and validating requirements: Tool support. In *Proceedings of the 19th International Conference on Software Engineering (ICSE'97)*.

Holten, D. (2006). Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5).

IEC (1998). Functional safety of electrical/electronic/programmable electronic safety-related systems. IEC 61508, International Electrotechnical Commission.

ISO (2011). Road vehicles – Functional safety. ISO 26262, International Organization for Standardization, Geneva, Switzerland.

ISO/IEC (2001). Software engineering – Product quality. ISO/IEC 9126, International Organization for Standardization, Geneva, Switzerland.

ISO/IEC/IEEE (2011a). Systems and software engineering – Architecture description. ISO/IEC/IEEE 42010:2011(E), International Organization for Standardization, Geneva, Switzerland.

ISO/IEC/IEEE (2011b). Systems and software engineering – Life cycle processes – Requirements engineering. ISO/IEC/IEEE 29148:2011(E), International Organization for Standardization, Geneva, Switzerland.

ITU-T (2011). Formal description techniques – Message sequence chart (MSC). ITU standard, International Telecommunication Union (Standardization Sector).

Jackson, D. (2002). Alloy: A lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2).

Jackson, M. and Zave, P. (1995). Deriving specifications from requirements: An example. In *Proceedings of the 17th International Conference on Software Engineering (ICSE'95)*.

Jackson, M. and Zave, P. (1998). Distributed feature composition: A virtual architecture for telecommunications services. *IEEE Transactions on Software Engineering*, 24(10).

Jarke, M., Bui, X. T., and Carroll, J. M. (1998). Scenario management: An interdisciplinary approach. *Requirements Engineering*, 3(3-4).

Jones, T. C. (1998). *Estimating software costs*. McGraw-Hill, Inc., Hightstown, NJ, USA.

Junker, M. and Neubeck, P. (2012). A rigorous approach to availability modeling. In *Proceedings of the 4th International Workshop on Modeling in Software Engineering (MISE'12)*.

Kang, K. C., Cohen, S., Hess, J., Novak, W., and Peterson, S. (1990). Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University.

Kang, K. C., Kim, S., Lee, J., Kim, K., Shin, E., and Huh, M. (1998). FORM: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5(1).

Kang, K. C., Lee, J., and Donohoe, P. (2002). Feature-oriented project line engineering. *IEEE Software*, 19(4).

Kästner, C., Apel, S., ur Rahman, S. S., Rosenmüller, M., Batory, D., and Saake, G. (2009). On the impact of the optional feature problem: Analysis and case studies. In *Proceedings of the 13th International Software Product Line Conference (SPLC'09)*.

Kazmierczak, E., Winikoff, M., and Dart, P. (1998). Verifying model oriented specifications through animation. In *Proceedings of the 5th Asia Pacific Software Engineering Conference (APSEC'98)*.

Keller, R. M. (1976). Formal verification of parallel programs. *Communications of the ACM*, 19(7).

Kelly, T. and Weaver, R. (2004). The goal structuring notation – A safety argument notation. In *Proceedings of the 1st Workshop on Assurance Cases*.

Kof, L. and Penzenstadler, B. (2011). From requirements to models: Feedback generation as a result of formalization. In *Advanced Information Systems Engineering*, volume 6741 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg.

Kugele, S. (2012). *Model-Based Development of Software-intensive Automotive Systems*. PhD thesis, Technische Universität München.

Landis, J. R. and Koch, G. G. (1977). The measurement of observer agreement for categorical data. *Biometrics*, 33(1).

Leveson, N. G. (1995). *Safeware: System Safety and Computers*. Addison-Wesley.

Liu, J., Batory, D., and Lengauer, C. (2006). Feature oriented refactoring of legacy applications. In *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*.

Lucassen, G., van der Werf, J., and Brinkkemper, S. (2014). Alignment of software product management and software architecture with discussion models. In *Proceedings of the 8th International Workshop on Software Product Management (IWSPM'14)*.

Luisa, M., Mariangela, F., and Pierluigi, I. (2004). Market research for requirements analysis using linguistic tools. *Requirements Engineering*, 9(1).

Méndez Fernández, D., Penzenstadler, B., Kuhrmann, M., and Broy, M. (2010). A meta model for artefact-orientation: Fundamentals and lessons learned in requirements engineering. In *Model Driven Engineering Languages and Systems*, volume 6395 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg.

Meyer, B. (1992). Applying "design by contract". *IEEE Computer*, 25(10).

Misra, S., Kumar, V., and Kumar, U. (2005). Goal-oriented or scenario-based requirements engineering technique – What should a practitioner select? In *Proceedings of the 18th Canadian Conference on Electrical and Computer Engineering (CCECE'05)*.

Mou, D. and Ratiu, D. (2012). Binding requirements and component architecture by using model-based test-driven development. In *Proceedings of the 1st IEEE International Workshop on the Twin Peaks of Requirements and Architecture (TwinPeaks'12)*.

Neubeck, P. (2012). *A Probabilistic Theory of Interactive Systems*. PhD thesis, Technische Universität München.

OMG (2011). OMG unified modeling language (OMG UML), superstructure. Version 2.4.1, Object Management Group.

OMG (2012). OMG systems modeling language (OMG SysML). Version 1.3, Object Management Group.

Onwuegbuzie, A. and Leech, N. (2007). Validity and qualitative research: An oxymoron? *Quality & Quantity*, 41(2).

Parnas, D. L., Madey, J., and Iglewski, M. (1994). Precise documentation of well-structured programs. *IEEE Transactions on Software Engineering*, 20.

Penzenstadler, B. (2011). *DeSyRe – Decomposition of Systems and their Requirements*. PhD thesis, Technische Universität München.

Pohl, K. (2010). *Requirements Engineering: Fundamentals, Principles, and Techniques*. Springer Publishing Company, Incorporated, 1st edition.

Pohl, K., Böckle, G., and van der Linden, F. J. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag.

Post, A., Menzel, I., Hoenicke, J., and Podelski, A. (2012). Automotive behavioral requirements expressed in a specification pattern system: A case study at BOSCH. *Requirements Engineering*, 17(1).

Prehofer, C. (1997). Feature-oriented programming: A fresh look at objects. In *ECOOP'97 – Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg.

Prehofer, C. (2013). Behavioral refinement and compatibility of statechart extensions. In *Proceedings of the 9th International Workshop on Formal Engineering Approaches to Software Components and Architectures (FESCA'13)*.

Prenninger, W. and Pretschner, A. (2005). Abstractions for model-based testing. *Electronic Notes in Theoretical Computer Science*, 116(0).

Pretschner, A., Broy, M., Krüger, I. H., and Stauner, T. (2007). Software engineering for automotive systems: A roadmap. In *Proceedings of the International Conference on the Future of Software Engineering (FOSE'07)*.

Robertson, S. and Robertson, J. (1999). *Mastering the requirements process*. Harlow, UK: Addison Wesley.

Robinson-Mallett, C. (2012). An approach on integrating models and textual specifications. In *Proceedings of the 2nd IEEE Model-Driven Requirements Engineering Workshop (MoDRE'12)*.

Sawant, K. P., Roy, S., Sripathi, S., Plesse, F., and Sajeev, A. S. M. (2014). Deriving requirements model from textual use cases. In *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*.

Schätz, B. (2008). Modular functional descriptions. *Electronic Notes in Theoretical Computer Science*, 215.

Schwaber, K. and Beedle, M. (2001). *Agile Software Development with Scrum*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Shaker, P., Atlee, J., and Wang, S. (2012). A feature-oriented requirements modelling language. In *Proceedings of the 20th IEEE International Requirements Engineering Conference (RE'12)*.

Siegmund, N., Kolesnikov, S. S., Kästner, C., Apel, S., Batory, D., Rosenmüller, M., and Saake, G. (2012). Predicting performance via automated feature-interaction detection. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*.

Sommerville, I. (2011). *Software Engineering*. Pearson Education, 9th edition.

Sutcliffe, A. (2003). Scenario-based requirements engineering. In *Proceedings of the 11th IEEE International Requirements Engineering Conference (RE'03)*.

Sutcliffe, A., Maiden, N. A. M., Minocha, S., and Manuel, D. (1998). Supporting scenario-based requirements engineering. *IEEE Transactions on Software Engineering*, 24(12).

Tarr, P., Ossher, H., Harrison, W., and Sutton, Jr., S. M. (1999). N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*.

Taylor, R. N., Medvidovic, N., and Dashofy, E. M. (2009). *Software architecture: Foundations, theory, and practice*. Wiley Publishing.

Teufl, S., Böhm, W., and Pinger, R. (2014). Understanding and closing the gap between requirements on system and subsystem level. In *Proceedings of the 4th International Model-Driven Requirements Engineering Workshop (MoDRE'14)*.

Teufl, S., Mou, D., and Ratiu, D. (2013). MIRA: A tooling-framework to experiment with model-based requirements engineering. In *Proceedings of the 21st IEEE International Requirements Engineering Conference (RE'13)*.

Thüm, T., Schaefer, I., Kuhlemann, M., Apel, S., and Saake, G. (2012). Applying design by contract to feature-oriented programming. In *Fundamental Approaches to Software Engineering*, volume 7212 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg.

Tran, L. M. S. and Massacci, F. (2014). An approach for decision support on the uncertainty in feature model evolution. In *Proceedings of the 22nd IEEE International Requirements Engineering Conference (RE'14)*.

Troubitsyna, E. (2008). Elicitation and specification of safety requirements. In *Proceedings of the 3rd International Conference on Systems (ICONS'08)*.

van Lamsweerde, A. (2001). Goal-oriented requirements engineering: A guided tour. In *Proceedings of the 5th IEEE International Symposium on Requirements Engineering (RE'01)*.

van Lamsweerde, A. (2009). *Requirements Engineering: From system goals to UML models to software specifications*. John Wiley & Sons.

Vogelsang, A. (2014). An exploratory study on improving automotive function specifications. In *Proceedings of the 2nd International Workshop on Conducting Empirical Studies in Industry (CESI'14)*.

Vogelsang, A., Eder, S., Hackenberg, G., Junker, M., and Teufl, S. (2014). Supporting concurrent development of requirements and architecture: A model-based approach. In *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD'14)*.

Vogelsang, A., Femmer, H., and Winkler, C. (2015). Systematic elicitation of mode models for multifunctional systems. In *Proceedings of the 23rd IEEE International Requirements Engineering Conference (RE'15)*.

Vogelsang, A. and Fuhrmann, S. (2013). Why feature dependencies challenge the requirements engineering of automotive systems: An empirical study. In *Proceedings of the 21st IEEE International Requirements Engineering Conference (RE'13)*.

Vogelsang, A., Teuchert, S., and Girard, J. (2012). Extent and characteristics of dependencies between vehicle functions in automotive software systems. In *Proceedings of the 4th International Workshop on Modeling in Software Engineering (MISE'12)*.

Voss, S. and Schätz, B. (2013). Deployment and scheduling synthesis for mixed-critical shared-memory applications. In *Proceedings of the 20th IEEE International Conference and Workshops on Engineering of Computer Based Systems (ECBS'13)*.

Wagner, S., Schätz, B., Puchner, S., and Kock, P. (2010). A case study on safety cases in the automotive domain: Modules, patterns, and models. In *Proceedings of the 21st International Symposium on Software Reliability Engineering (ISSRE'10)*.

Waldmann, B. and Jones, P. (2009). Feature-oriented requirements satisfy needs for reuse and systems view. In *Proceedings of the 17th IEEE International Requirements Engineering Conference (RE'09)*.

West, M. and Eaglestone, B. (1992). Software development: Two approaches to animation of Z specifications using Prolog. *Software Engineering Journal*, 7(4).

Yu, E., Giorgini, P., Maiden, N., and Mylopoulos, J. (2011). *Social Modeling for Requirements Engineering*. The MIT Press.

Zave, P. (1999). FAQ sheet on feature interaction. `http://www.research.att.com/~pamela/faq.html`.

Zave, P. (2001). Requirements for evolving systems: A telecommunications perspective. In *Proceedings of the 5th IEEE International Symposium on Requirements Engineering (RE'01)*.

Zave, P. (2003). An experiment in feature engineering. In *Programming methodology*. Springer.

Zave, P., Goguen, H. H., and Smith, T. M. (2004). Component coordination: A telecommunication case study. *Computer Networks*, 45(5).