Specification and Analysis of Availability for Software-Intensive Systems

Maximilian Junker

Technische Universität München

Institut für Informatik
der Technischen Universität München

Specification and Analysis of Availability for Software-Intensive Systems

Maximilian Christian Junker

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. Arndt Bode
Prüfer der Dissertation:		
	1.	Univ.-Prof. Dr. Dr. h.c. Manfred Broy
	2.	Univ.-Prof. Dr. Ralf Reussner,
		Karlsruher Institut für Technologie

Die Dissertation wurde am 10.05.2016 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 05.09.2016 angenommen.

Abstract

For many technical systems, their availability is an important characteristic. Examples for systems, where availability is especially relevant, are railway control systems, telecommunication systems, and systems supporting business processes. Outages of such systems often have immediate economic consequences for their users and operators. Due to the economic impact, manufacturers and service providers need to guarantee a certain level of availability and these guarantees become part of the system requirements.

To avoid high costs due to changes in late development phases, availability requirements need to be specified and verified as early as possible. To enable early predictions of the system availability, a range of modeling techniques have been proposed by academia and are in use in practice. However, current modeling techniques have two drawbacks: First, they describe the system on a high level of abstraction, usually in terms of their architecture. This makes it hard to formulate meaningful, system-specific availability requirements that relate to the functional requirements of the system. Second, the current techniques are not embedded into a comprehensive engineering method, which defines relationships between different models and provides an engineering process.

In this thesis, we provide evidence for the stated problems and propose a solution. First, we report on a qualitative interview study we conducted with 15 industrial availability experts from different domains. With this study, we assess the relevance of the topic availability in the industry, determine availability related activities and methods, and identify several problems connected to the specification and verification of availability requirements.

Second, as a solution to the stated problems, we extend an existing artifact model for software-intensive systems by additional artifacts supporting the concise specification and analysis of system-specific availability properties. The first additional artifact is an availability requirements specification. It uses concepts from the second artifact, the availability specification, which captures system-specific definitions of failure and availability metrics. The third artifact is an extended logical architecture that includes the system's behavior in case of faults. The last artifact is an environment specification, which contains the structure and behavior of the system's environment. For each additional artifact, we suggest suitable models and description techniques to capture the necessary information.

Third, we provide a modeling method that supports the systematic application of our artifact model. The modeling method consists of basic modeling building blocks, a process for instantiating the artifact model, step-by-step guides for systematically creating individual models, and modeling patterns providing a basic structure for some of the model types.

Finally, we evaluate our artifact method and our modeling method in an industrial case study. In the case study, we assess the adequacy and the flexibility of our modeling approach. We further report on prototypical tool support and evaluate the feasibility of an availability analysis based on the created models. To perform the case study, we model a section of an industrial train control system, departing from the original requirements. We then extend this initial model by instantiating our availability artifact model. As a last step, we perform several types of availability analyses.

Acknowledgements

This work would not have been possible without the support by various people. In particular I would like to thank Prof. Manfred Broy for giving me the opportunity to work in his research group. This group is exceptional in the way it combines basic and industrial research and this thesis greatly profited from this combination. I also thank Manfred Broy for the various fruitful discussions about system modeling and availability. For reviewing the thesis and providing feedback, I also thank Prof. Ralf Reussner.

During the last five years I had the luck to work with many great colleagues. I would like to thank all of them. In no particular order, I would like to especially thank Andreas Vogelsang for the many discussions and feedback on this thesis, Veronika Bauer for helping me with the interview study, Benedikt Hauptmann for being a great office roommate, for his great suggestions on thesis writing and for feedback on various parts of this thesis, Sebastian Eder for helping me with the automated analyses infrastructure, Wolfang Böhm for providing feedback on my writings and for organizing the project with Siemens, Johannes Hölzl for providing feedback on the mathematical foundation, and Philipp Neubeck for explaining me his probabilistic system modeling theory.

Working closely with the industry (especially Siemens) helped me to understand the problems associated with availability modeling and analysis and to evaluate my approach on a real-life example. For this possibility and the support during our cooperation I would like to thank Ralf Pinger, Ernesto de Stefano and Karsten Rahn, all of them from Siemens in Braunschweig. I also thank all industry partners that agreed to be interviewed for my study on the state of the practice.

Finally, but most importantly, I would like to thank my parents and my sister Constanze for the ongoing support for the last 32 years, my friends for being my friends and Fidi, for listening and encouraging me many many times.

Contents

Chapter 1

Introduction

In our daily life, we are surrounded by technical systems that rely to a great extent on software to provide their service. We demand from these software-intensive systems to be operating when we need them. For instance, we expect a phone call to be put through, when we call a number, and a train to be in service according to the timetable. However, all technical systems can fail and can therefore be out of service for some time. The share of time when a system is operating, is called its availability. The topic of this thesis is the model-based specification and analysis of availability for software-intensive system. In this chapter, we first motivate the topic in Section 1.1. Afterwards, we state a number of currently open problems in this field in Section 1.2. We present an overview over our suggested approach in Section 1.3 and the contributions of this thesis in Section 1.4. Finally, we give an outline of this thesis in Section 1.5

1.1 Context: Availability of Software-Intensive Systems

A *software-intensive system* is "any system where software contributes essential influences to the design, construction, deployment, and evolution of the system as a whole" (IEEE, 2007). Other than for pure software systems, however, the hardware cannot be neglected during the development. The reason for this is that the hardware still contributes significantly to the behavior of the system, e.g. through hardware implemented functionality or because it can fail due to wear out. Software-intensive systems are pervasive: Many services and applications that we use regularly are controlled by software-intensive systems. Examples are the telephone system, subway trains or the cashing system at supermarkets. These examples show that software-intensive systems are often complex systems. More specifically, they are large, often distributed and typically provide several functions.

For many of these software-intensive systems, their *availability* is an important property. A highly available system is operating failure-free most of the time. On the contrary a system with a poor availability is often out of service for long periods of time. Fields, where availability is especially important, are industrial systems for telecommunication, transportation or production, as well as business information systems. Inadequate availability of such systems can have an immediate economic impact for customers (such as a train operator) as well as for manufacturers. For instance, in the case of a train operator, insufficient availability of a train control system produces unplanned costs, including penalties, lost earnings, lost reputation, lost worker hours and recovery costs. In other cases, warranty and liability costs may add to this (IEC, 2004). For a manufacturer, it is important to deliver a product with the right level of availability. Too high availability may result in the increase of costs and the decrease of competitiveness, due to higher development efforts and more expensive equipment.

Specifying the required availability and verifying these requirements as early as possible during the development of a software-intensive system, is highly desirable to avoid costs for re-designing the system. In practice, extensive availability analyses are performed based on the architecture of the systems to prove that the system fulfills the availability requirements. However, these analyses often involve a large fraction of manual work for understanding the impact of faults on the behavior of the system. Furthermore, the analyses are often rather generic. For example, they often do not differentiate between the availability of different system functions. Furthermore, the basis of the analyses is often a notion of availability that is not specific for the given system. This results in over- or underestimating the system's availability and can lead to disputes with the customer or the need to redo parts of the analysis.

1.2 Problem Statement

To support the specification and analysis of availability properties in software-intensive systems, model-based techniques have been proposed by research. These techniques usually capture the system architecture and the way that faults propagate to the system interface. These models can be analyzed automatically and thus relieve the engineers from parts of the manual work for availability analysis. However, several problems are not solved by the model-based approaches proposed today. In this thesis, we address the following two problems.

Problem 1: Formulating and Analyzing System Specific Availability Requirements

In an interview study with participants from industry, presented in Chapter 2, we found that formulating precise availability requirements is a problem today. Availability requirements need to state explicitly, which types of failures should be considered, which should be excluded, and how availability metrics should be calculated. Availability requirements, such as "the weekly downtime should be no more than 1 hour", do not precisely state which kinds of failure are included and how uptime should be exactly calculated. They thus stay ambiguous. Such requirements furthermore do not relate to the functional requirements of a system and it remains unclear which functions or functional requirements need to be violated to which degree to consider a system as unavailable. Such vague requirements are hard to interpret and verify, and are a source for misunderstanding between manufactures and customers. Current availability specification and analysis approaches allow to formulate availability requirements only by giving availability metric values and do not support system specific definitions of failure and availability metrics.

Problem 2: Lack of Integration into an Engineering Method

In order to employ an availability modeling and analysis approach in practice, it needs to be integrated into a comprehensive engineering method. First, such an integration includes a description of how the availability specific models relate to other models (e.g., of the requirements or the architecture). This allows to identify models that need to be created beforehand and to maintain the availability models in case the functional requirements or the architecture changes.

Second, the creation of availability models needs to be integrated into a process, describing how the different models are sequentially developed. The current availability modeling approaches lack such a comprehensive integration, which hinders their adoption.

1.3 Approach

Our approach to the problem of specifying and analyzing availability properties is to extend an existing artifact model for model-based engineering of software-intensive systems with specific models for availability. The main purpose of the additional models is to define, what availability means for the system under development. Most importantly, they specify which deviations of the observed behavior from the specified behavior should be considered as which type of failure and how availability metrics should be calculated based on these specific failure definitions. Additional models describe the effect of faults on the behavior of system components and the system environment.

Our modeling approach allows us to specify availability requirements that have a precise interpretation in terms of the system behavior and a clear relation to the system's functional requirements. Even complex availability conditions can be modeled, as long as they can be expressed in terms of the system behavior. Our approach further enables a completely automated analysis of the system availability. This addresses the general problem of labor intensive and error prone manual availability analysis. Moreover, our added models do not duplicate any information already contained in existing models. Instead, they only add new information regarding the system specific definition of availability, faults, and the environment.

1.4 Contributions

In this thesis, we provide the following four contributions to motivate, implement and validate the above approach.

Interview Study on State of the Practice We present an interview study with participants from the industry that examines the state of the practice regarding availability engineering. Most importantly, it uncovers several problems present in the industrial practice.

Availability Artifact Model We suggest an artifact model that includes models for availability. The artifact model extends an existing artifact model for model-based requirements engineering. We further show, how an availability analysis can be performed using our proposed set of models.

Availability Modeling Method We provide a method for systematically creating these availability models. The method consists of a modeling process, basic building blocks, modeling patterns and step-by-step guides. The method provides guidance for creating availability models and thus helps to employ our approach in practice.

Case Study: Train Door Control To evaluate our approach, we apply it to an industrial case example. The case example consists of the train door control part of a commercial train control system. In the case study, we follow our method for creating availability models and perform a detailed availability analysis with a prototypical tool.

1.4.1 Study on the State of the Practice

We present a qualitative interview study with 15 availability experts from industry, representing various domains such as railway, automation and business information systems. In this study, we investigate the relevance of availability in these domains. Furthermore, the study describes which availability related activities are carried out and which problems the experts from industry perceive with respect to availability engineering. Main results of the study are:

- Availability is of different relevance for different types of companies and in different domains, depending for example on industry standards and regulations, the type of customers or economic considerations.
- Availability related activities are carried out in all phases of the development lifecycle from requirements engineering until system operation and maintenance.
- The experts report on several difficulties regarding, for example, requirements formulation and design-time availability analysis. Moreover, availability considerations pose inherent challenges, such as obtaining needed spare parts over a long period of time.

1.4.2 Availability Artifact Model

We extend an existing artifact model for model-based requirement engineering with artifacts and models for availability. The models are based on the formal modeling theory Focus (Broy and Stølen, 2001; Neubeck, 2012). We introduce the following four additional artifacts.

Availability Requirements Specification The availability requirements specification captures the demands of the different stakeholders that relate to availability. We capture availability requirements informally with *textual availability descriptions* and formalize these descriptions in an *availability constraints model*. The latter model represents availability requirements as mathematical structures that relate to well-defined, formal availability metrics as specified in the availability specification artifact.

Availability Specification The availability specification defines what availability should mean for the system under consideration. More specific, its *failure definition model* contains a definition of what failure means. Its *availability metric model* specifies how to calculate availability metrics. In combination, these models provide the basis, on which formal availability requirements are formulated.

Extended Logical Architecture The purpose of the extended logical architecture is to include the behavior of the system in case of faults. Only when faults and their effect on the system behavior are modeled, we can perform meaningful availability analyses. The only model in this artifact is the *fault-injection model*. This model is a description of how components are affected by faults and how these faults lead to a change in behavior.

Environment Specification An environment specification describes the context of a system, for example, its neighboring systems, its physical environment, or its users. Its purpose is to allow for a realistic availability analyses, as a system may show a different availability depending on its use.

Based on the suggested models we describe a formal availability analysis to verify availability requirements.

1.4.3 Availability Modeling Method

Our third contribution is a method for systematically developing availability models according to our artifact model. The method contains the following elements

Basic Building Blocks We provide 19 generic, parametrizable specifications for availability models. They include specifications for availability metrics, fault injection and behavior comparison.

Process The artifact model does not prescribe a fixed process. We suggest one specific process for the sequential creation of availability models.

Step-by-step Guides For some of the individual models we provide step-by-step guides for their development. As an example, we suggest a guide word based approach for the elicitation and documentation of failure modes.

Modeling Patterns We introduce modeling patterns to provide a basic model structure and thus ease the creation of availability models.

We illustrate the method using a running example of a storage and access system.

1.4.4 Case Study: Train Door Control

The last contribution is a case study to evaluate our modeling approach. The case example is an industrial train control system developed by the company Siemens. To obtain input for availability modeling, we performed interviews with Siemens engineers. With this information, we were able to create availability models according to our artifact model and modeling method. We found, that we could capture most of the availability requirements with our models. We used the models to perform a series of availability analyses, verifying the requirements and comparing architecture alternatives with respect to availability. To execute the analyses automatically, we created prototypical tool support.

1.5 Outline

The main part of this thesis is structured in the following seven chapters. In Chapter 2, we report on the interview study we performed with experts from different domains of the industry in order to investigate the state of the practice regarding availability. This study on the state of the practice is complemented with a review on the state of the art in Chapter 3. In Chapter 4, we review basic mathematical concepts and the Focus modeling theory, on which all of our suggested models are based. Moreover, this chapter discusses availability and related notions. In Chapter 5, we present our artifact model and a formal treatment of availability analysis. In Chapter 6, we introduce the availability modeling method. Chapter 7 presents an industrial case study where we evaluate the applicability of the outlined approach. Finally, Chapter 8 discusses opportunities for further research and summarizes the thesis.

Previously Published Material

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

(Junker and Neubeck, 2012) Maximilian Junker and Philipp Neubeck: A Rigorous Approach to Availability Modeling, in: 2012 Workshop on Modeling in Software Engineering (MISE)

(Böhm et al., 2014) Wolfgang Böhm, Maximilian Junker, Andreas Vogelsang, Sabine Teufl, Ralf Pinger and Karsten Rahn: A formal systems engineering approach in practice: An experience report, in: Proceedings of the 1st International Workshop on Software Engineering Research and Industrial Practices, 2014

(Junker, 2014) Maximilian Junker: Exploiting Behavior Models for Availability Analysis of Interactive Systems, in: Proceedings of the 2014 International Symposium on Software Reliability Engineering

Chapter 2

Availability: State of the Practice

Availability is a topic relevant for many industries. However, there is no first-hand information on how availability is actually understood and handled in practice and what problems practitioners are facing. To acquire this information, we performed a qualitative interview study with availability experts from several branches of the industry that are concerned with software-intensive systems. In this chapter, we report on this study. We first formally state the study goal and the research questions that guided the interview design in Section 2.1. Afterwards, we give details on the interview design and the execution of the study in Section 2.2. Section 2.3 presents the results of the study. We discuss our findings in Section 2.4. The rest of the chapter addresses threats to validity and gives a conclusion.

2.1 Study Goals and Research Questions

In order to obtain a systematic research process we formulate a study goal from which we derive four research questions. We use the research questions in a next step to create a first interview guideline.

2.1.1 Study Goal

We formally state our study goal using a slightly adapted version of a goal definition template due to Wohlin et al. (2012).

We analyze	*the state of the practice regarding availability*
for the purpose of	*understanding it and identifying existing problems*
from the viewpoint of	*availability experts*
in the context of	*industrial companies from different domains developing software intensive systems.*

2.1.2 Research Questions

In order to reach our goal, we devised the following four research questions.

RQ 1: Relevance – How relevant is the topic availability in the industry?

With Research Question 1, we want to investigate how the industry assesses the relevance of availability. A low relevance would be a hint that this line of research is not worthwhile or that the expert selection was not adequate. Furthermore, we are interested in the reasons why availability is seen as relevant by the experts. Last, we want to know, if there are differences in relevance related to the domain or specific context.

RQ 2: Understanding – What understanding of availability is present in the industry?

With Research Question 2, we want to gain an insight into what notions of availability are present in the industry and how this notion varies among different companies. There are two relevant subtopics which we consider separately. The first topic is the question, under which condition a system is considered unavailable. This relates to the notion of failure in the context of availability. The second topic then relates to the actual notion of availability as a system property. In order to distinguish those topics we split RQ 2 into two subquestions:

RQ 2a: Understanding of Failure. Here, we want to investigate the criteria when a system is considered as failed. What aspects of the system are considered relevant in order to decide this and how are failures described?

RQ 2b: Understanding of Availability. Based on the understanding of failure, how is the concept availability understood? Which metrics are used, if any? What are other means to capture availability?

RQ 3: Activities – What activities are performed during the whole product life-cycle that are related to the availability of a system?

With Research Question 3, we aim to identify the levers that are used in practice to achieve the availability goals. This question is not only geared towards system architecture development, but encompasses the whole product life-cycle including, for example, project management, requirements engineering, system operation and maintenance.

RQ 4: Problems – What problems are perceived regarding availability?

With Research Question 4, we want to identify shortcomings in the current practice regarding the realization of highly available systems.

2.2 Study Method

In this section, we explain the design of the interview study. More specifically, we first describe how we selected the study participant in Section 2.2.1. In Section 2.2.2, we present the interview guideline we developed to structure the interviews. In Section 2.2.3, we outline how we actually conducted the interviews. Finally, in Section 2.2.4, we give details on how we analyzed the interview notes in order to answer our research questions.

#	Company	System Types	Expert Role
1	A	Train Protection Systems	RAM Specialist
2	A	Industrial Controller	Product Manager
3	A	Process Automation Systems	RAM Specialist
4	B	Intra-Logistics Automation Systems	Automation Engineer
5	B	Intra-Logistics Automation Systems	System Architect
6	C	Data Centers & Networks	System Architect
7	D	Database Systems	System Operation Engineer
8	D	Process Automation Systems	Automation Engineer
9	E	Railway Telematics Systems	System Architect
10	E	Train Protection Systems	System Architect
11	F	Business Information Systems	System Operation Engineer
12	F	Business Information Systems	System Operation Engineer
13	G	Miscellaneous	Software Architect
14	H	Miscellaneous	Software Architect
15	I	Business Information Systems	Software Architect

Table 2.1: Overview over the study participants.

2.2.1 Study Participants

In order to get a detailed picture of availability in industry, we decided to perform interviews with experts from a broad range of domains and company types. When selecting the companies we restricted ourselves to domains, where we already knew, or at least supposed, that availability could play a role. Such domains are telecommunication and transportation, but also, for instance, insurance. The companies and experts were selected in a way to achieve a diversity regarding the domain, business type (e.g. product or service) and size of the company, as well as the role of the department and the expert. We also tried to vary the types of systems that the experts are concerned with. For example, we included embedded systems, business information systems and infrastructure systems such as data centers. Moreover, we considered both, centralized and distributed systems. As sampling method we used a mixture of convenience sampling and snowball sampling (Kitchenham and Pfleeger, 2008) in the following way. We first addressed companies to which we had contact before and asked for experts that we can interview. Second, we selectively contacted additional companies to achieve more variance. Third, in each interview we asked the experts if they know of further persons that we should interview. We aimed to include experts that fulfill different roles in their organization in order to consider different views on availability. We therefore interviewed RAM specialists, whose tasks include availability analysis, software and system architects, automation engineers, whose task it is to plan automation systems, operation engineers, who are responsible for the operation of (software) systems, as well as a product manager. Table 2.1 gives an overview over the participants' roles and the systems types that they attend to.

2.2.2 Interview Guideline

We developed a guideline to structure the interviews. We first derived an initial interview guideline from the research questions. This guideline was adapted during the interviewing

Question Block	**Targeted RQs**
General	RQ 1 (Relevance)
Relevance of Availability	RQ 1 (Relevance)
Understanding of Availability	RQ 2 (Understanding)
Availability in the Product Lifecycle	RQ 2 (Understanding), RQ 3 (Activities)
Experiences	RQ 3 (Activities), RQ 4 (Problems)
Closing	no explicit RQ targeted

Table 2.2: Targeted research questions for the question blocks.

phase to reflect the experience that we gained with the first interviews. The final interview guideline consisted of the following five question blocks with 28 questions in total.

1. *General* (3 questions): Questions, regarding the expert's department, the role of the expert, as well as the systems that are considered. Questions in this block do not relate to a specific research question but instead are used as introduction and to understand the context of the expert.

2. *Relevance of Availability* (2 questions): Questions, regarding the relevance of availability in the company. Here, we specifically asked for the importance of availability and inquired situations, factors and systems for which availability is especially significant. We also asked for availability related standards and norms.

3. *Understanding of Availability* (3 questions): Questions, regarding the notions of failure and availability that the experts employ. Additionally, one question is related to the role of degradation and operating modes.

4. *Availability in the Product Lifecycle* (15 questions): Questions, regarding the role of availability with respect to the tasks project management, requirements engineering, architecture, implementation, verification, maintenance and operation. Additionally, we asked for availability related modeling activities that are performed.

5. *Experiences* (2 questions): Questions, regarding concrete projects or examples where availability played a major role. We asked for challenges and problems, their root causes, solutions and open issues.

6. *Closing* (3 questions): Questions, regarding further points that should be discussed and further interview partners. Additionally, we asked directly what an ideal modeling technique for availability should contain.

A complete list of all questions in the interview guideline can be found in Appendix A. Table 2.2 shows how the question blocks relate to our research questions.

2.2.3 Interview Execution

We conducted the interviews either per telephone or in person. Each interview took between 30 minutes and two hours. Usually, we only interviewed one person at a time. Only in two cases we interviewed a group of persons. We performed each interview with two interviewers, one responsible for performing the actual interview and one for creating a protocol. During the interviews, the guideline was mainly used as a checklist in order to ensure that we cover the important topics. In some cases, we also discovered a new topic during an interview. We added such a topic to the guideline after the interview if we thought it could be relevant in other cases, too.

2.2.4 Interview Analysis

The interview protocols were the source for the analysis. For each of our four research questions, we identified the main topics that were mentioned by the experts and grouped the statements of the experts according to these topics. We did not use the interview guideline to structure the protocols, as not all aspects of the guideline were covered in each interview and furthermore in some interviews specific topics came up, which were not part of the guideline. Apart from sorting the statements into topics, we also assigned one or more keywords to each statement. We used this structured representation of the protocols to answer our research questions. The organization and analysis of the protocols were performed using a spreadsheet tool.

2.3 Results

In the following, we present the results of the interview study, according to our research questions. We number the topics that emerged in the interviews by $T\{Number\}$, in order to refer to the topics in the discussion.

2.3.1 RQ 1: Relevance

During our interviews we found four major themes regarding the relevance of availability.

Availability as Industry-specific Standard (T1). In some industries (automation, medical, telecommunication), a certain availability level is either customary in this industry branch or it is demanded by industry standards. For example, one expert from industry automation stated that a minimum availability according to FEM (European Materials handling Federation) always becomes part of the contractual basis. A different expert mentioned guidelines by certification organizations for medical equipment such as the FDA (US Food and Drug Administration). According to the expert, these kinds of guidelines are increasingly demanding high availability, additionally to safety which has traditionally been emphasized. An expert from the telecommunication domain referred to country-specific laws for suppliers of critical infrastructure, which demand certain levels of availability.

Availability as a Customer Requirement (T2). In several interviews the experts stated that they need to fulfill explicit customer requirements regarding availability. In the rail and industry automation domains, the experts said they need to provide detailed argumentation that a certain availability level is realized in the system. This is typically done in the form of a RAM case, a document that contains such kind of argumentation in a structured way. In other cases, certain guaranteed availability metrics are part of the contract for a system, e.g. as part of a Service-level Agreement.

Availability as Quality Indicator (T3). Availability is not always a hard requirement but sometimes rather a quality indicator. That means, there is no binding specification of availability that needs to be reached, or this specification is only for internal quality control. This is stated by an expert involved in railway telematics systems, who reported that there exist internal availability goals, which are, however, not quantitatively verified. This topic primarily emerged in cases, where the systems are provided to internal customers. It does not imply that availability is not important in these situations. The level of availability may even be systematically measured but there is no compulsory proof of availability. In such a context, the systems are often designed to be as available as possible within the given budget. For instance, two experts with such a background told us that they always pursue a "best-effort" strategy regarding availability.

Economical Trade-off (T4). Across different domains, interviewees emphasized the fact that availability is to a large extent driven by economic factors. The customers of a system need to decide which level of availability with associated costs is economically sensible. As one expert put it: "Availability is always a trade-off. The 100% solution is too expensive".

2.3.2 RQ 2a: Understanding – Failure

With this research question, we wanted to explore the experts' notion of a failure in the context of availability. We extracted four dimensions that describe the bandwidth of the different understandings of failure present in our interviews.

Intuitive Understanding vs. Precise Definition (T5). In several cases, the interviewees stated that they do not employ a formal definition of failure. Instead, an intuitive definition is used. An example for such an intuitive understanding is the statement: "Every deviation from the nominal state is a failure". One expert from the business information system domain acknowledged that, especially for complex system, an intuitive notion of failure is employed, which, at the same time, is limited in expressiveness. On the other hand, there were also cases where rather precise definitions of failure existed, such as thresholds for correct values of a functionality together with time-bounds. Such precise definitions existed for example in the medical domain. In this domain, we observed a strong relationship between availability and safety, as the unavailability of a certain function may be a safety hazard.

Scope (T6). Experts from different domains considered failures with respect to different scopes. With scope, we relate to the subject-matter under consideration. In a hierarchical system this could be the system itself or some sub-system or sub-functionality. For instance, in the case of control systems for process automation, failures are primarily defined with respect to the scope of a plant or a part of the plant, such as a pump. In this example, a failure of the control system is defined in terms of the performance of the pump. A further example of a super system as scope for failure is the rail domain. Here, one expert stated that "outside visibility" of a failure is an important criteria. Outside visibility is given, for example, when a train gathers more than 3 minutes delay due to this failure. However, in most interviews the scope of a failure was the system itself or a specific functionality of the system.

Viewpoint (T7). Experts understood failure either from a black-box (interface behavior) view or a glass-box (state space or internal structure) view. In the interviews, the experts took mainly an black-box view when relating to failures. The black-box view was apparent when they named incorrect outputs and delayed responses as examples for failures. Furthermore, metrics such as throughput or bandwidth relate to a black box view on the system. For instance, one expert from the data center domain stated that their main criterion is that the deployed services are be executable without delay.

However, in some cases also a glass-box view was present. A prominent example for failures defined with respect to a glass-box view, in this case regarding the system state, is data-loss. Experts from the datacenters and automation domains stated that data-loss is an important class of failure that they consider. The experts provided a number of examples that relate sub-systems (or sub-functions). Two experts explained that a system failure with respect to availability is determined by the system functions that are active at that moment. For instance, an expert from the medical systems domain described an infusion controlling system where the main functionality is protected by a safety function that is activated in case the main function cannot be performed anymore. In this case, the system is considered unavailable as soon as the safety function takes over. A similar example stems from the rail domain. Here, the availability of an automated driving function is determined by the set of sub-functions that do currently operate. In the automation domain, it is common to assign weights to sub-functions that describe the importance of sub-function failures for super-function failures.

Failure Classifications (T8). This dimension relates to the way failures are categorized. On a high level we can distinguish two cases: In the first case, only two categories “operational” and “failed” are considered. In the other case, several levels of failure are distinguished, e.g. regarding failure severity. Especially for complex, multi-functional systems, a binary notion of failure is often seen as inadequate. In many cases, the number of failure classes was small. Often, the experts reported about three to four classes. The experts reported on a number of criteria to make the distinction. A first criterion is the degree of degradation that a system exhibits. For example, an expert stated that in the context of web-based information systems, they use the following categorization: 1) The system behaves as expected 2) The system answers but the response is not as expected (e.g. a failure notice) 3) The system does not respond at all. How to categorize a concrete behavior, and especially how to distinguish between 1) and 2) in the case of small deviations from the specified behavior, is up to the engineer. The range of the disturbance was named as further criterion. As an example for the latter, an interviewee from the railway telematics domain stated that it makes a difference if only a few work-stations are out of service or a large number of then. Similar, a different expert employs a categorization based on the number of employees that are affected.

A third criteria is the time when the failure occurs. In two cases, experts reported that they have different kind of systems which are guaranteed to operate during different time periods. For example, the following classification was used: 1) Monday - Friday, 8am - 6pm, 2) Monday - Saturday, 6am - 8pm, 3) 24/7 . Depending on the system, where the failure occurs, and the day and time, failures are categorized in different classes.

2.3.3 RQ 2b: Understanding – Availability

Quantitative vs. Qualitative Properties (T9). Most interviewees understood availability as a measure for the amount of time when the system is not available (i.e. has failed). The two most prominent metrics mentioned were uptime (including and excluding planned maintenance) and repair-time. Two experts mentioned metrics that are not related to time. In one case this was diagnostic coverage, the probability that a fault is diagnosed. In the second case the expert referred to loss of money.

Although availability metrics were used broadly among the interviewees, many experts also criticized them for being not expressive and hard to measure. In some cases availability metrics were only used in marketing and to specify internal goals but not during the further development.

According to the experts, availability also served as an umbrella for a range of qualitative properties of a system regarding its failure behavior. An example for such a property is that the system must never exhibit data-loss. Similarly, one expert stated that because availability metrics are hard to handle they try to capture availability via “functional requirements”. An example for such a functional requirement, given by one of our experts, is failure disclosure, which relates to the ability of the system to issue a notification as soon as it has failed.

2.3.4 RQ 3: Activities

Project Management

Staffing (T10). With respect to project management, the interviewees named one main activity important for reaching availability goals. This strategy is to include availability specialists into the project team from an early phase on. Thus, early design decisions can be reviewed with respect to availability. An example stems from one of the interviewees, who has the role of a RAM specialist in the automation domain. He stated that in projects with explicit availability requirements, he is usually involved early, starting with the architecture development. A different expert from process automation stressed the importance of including external experts into a project, for instance experts affiliated with certification companies.

Requirements Engineering

With respect to requirements engineering, two important activities emerged in the interviews: Requirements elicitation (i.e. the extraction of the initial requirements from the stakeholder) and requirements analysis.

Requirements Elicitation (T11). The experts named two main types of sources for elicitation of availability requirements. They are in line with the types of relevance outlined in Section 2.3.1. The first source mentioned are the users and customers of the systems. One expert from process automation remarked that availability requirements elicitated from users tend to be unspecific: “The person in the operations room wants that the system is always running when he needs it”. In our interviews we encountered cases with external as well as internal customers. An expert from the business information systems domain experienced that customers, contrary to users, sometimes have rather specific availability requirements. This expert explained further that in cases where customer requirements are very specific they try to find out the reasons for those requirements (e.g. bad experiences in the past). A second source for availability requirements are industry standards and regulations. As mentioned before, there exist industry standards that impose certain availability requirements, for example FEM for automation or the FDA for medical devices.

Requirements Analysis (T12). All experts stressed the importance of economic and marketing considerations for availability. As, on the one hand, a high availability increases the value of a system but, on the other hand, high-availability always comes with a price, availability requirements are often the result of a cost-benefit analysis. An expert from the process automation domain acknowledged that view when stating that availability metrics often do not stem from (technical) experts but instead are motivated and derived from economic factors. Availability requirements may also emerge later in the process through derivation from other requirements or due to certain architectural decisions. Two experts reported that risk analyses and FMEAs (Failure Mode and Effect Analysis), which are carried out with respect to availability on the basis of the architecture, can lead to new requirements which address the issues that came up during the analysis. An expert from process automation explained that availability requirements for the automation system are derived from architectural decisions concerning the whole plant.

Architecture Development

A broad range of activities were mentioned during the interviews relating to the architecture of the system.

Creating Architecture Prototypes (T13). Architecture prototypes are used to evaluate the availability early in the design phase. An expert from the healthcare domain claimed that this strategy can avoid major architecture changes otherwise induced by availability problems.

Introducing Redundancy (T14). Often, the interviewees related to physical redundancy of some technical computing component (e.g. processor, memory, interface equipment, database). In case of a failure, the redundant unit can take over. A specific instance of redundancy is virtualization, where an application or a component can be moved between nodes of a highly redundant infrastructure. The experts also mentioned redundancy of infrastructure, such as energy supply, or networks. A further type of redundancy that was brought up, is data redundancy (e.g. through backups).

Choosing Components (T15). Additionally to redundancy, it is also possible to influence availability by choosing components for the architecture that are highly reliable themselves. One expert stated for instance, that for certain availability-relevant sub-systems, they employ only certified third-party software. Here, also long-term support and supply with spare-parts plays a role.

Introducing Fault Detection and Handling (16). Often, redundancy or highly reliable components are not enough to fulfill the availability requirements, or it is too expensive. Therefore, other means have to be introduced into the architecture. Failure isolation is used to prevent faults from propagating through a system. It is applied (on a system level) in the automation domain. Here, plants are deliberately designed in a way such that different functions of the plant fail independently. Thus a (degraded) operation of the plant is still possible. In case of a component fault, the system availability can sometimes be maintained for a while by the residual components compensating for the defective components. For example, a system may compensate for the failure of a database component by buffering requests to the database or by only providing old, cached data (which might be acceptable in certain situations). Moreover, only the subset of the functionality that does not need the database could be activated. An expert concerned with datacenters termed this kind of strategy “design for failure”. In all cases, monitoring of the system and its components is seen as crucial in order to be able to detect failures and be able to react.

Optimizing Deployment (T17). Deployment relates to the assignment of software to computing units and may have significant impact on the system’s availability. Hence, optimizing the deployment can increase the availability. Experts from the business information and from the industrial controller domains emphasized the importance of hot-deploy, which refers to the possibility to update the software on a system during operation.

Verifying Architecture w.r.t Availability (T18). One approach to verify that an architecture is able to provide a certain availability, is to perform manual reviews. One expert said, it is common in his company to perform workshops where architects and availability specialists review the current architecture. Apart from reviews, structured methods such as Failure Mode and Effects Analysis (FMEA) or an analysis based on failure scenarios were mentioned by the experts to investigate possible availability problems. The interviewees also named techniques based on architecture modeling. Prominent examples here are Failure Trees, Reliability Block Diagrams and Markov Models. A further technique, mentioned by these experts, is simulation. Often, simulation is performed on the basis of an architectural model. During the simulation, component faults are injected and the effects on the system are evaluated.

Software Development

As our focus is software intensive systems we explicitly asked for specific activities related to the software development. Compared to the activities regarding the architecture, the experts mentioned fewer points here.

Software Implementation (T19). Strategies with respect to software implementation mentioned by the experts includes the usage of reduced language sets and avoidance of memory access violations. One expert remarked that those techniques are, however, not specific to availability and can be considered good software engineering practice. A further aspect, mentioned by an expert from the automation domain, is sensible exception handling. The expert stressed the point that it is important to handle exceptional situations in a way that functionality that is not directly affected by the exception can continue to operate. He reported a case where the improper formatting of an input lead to a termination of the whole system. Instead, handling the situation by discarding the input would have been better from an availability point-of-view.

Testing (T20). Most experts stressed the importance of testing for availability. Especially integration tests and system tests are used for this purpose. An expert from the business information system domain explained that special tests are run in order to improve the availability. These tests verify, for example, that redundancy and fault-tolerance mechanisms are working or that backups can be correctly imported. Experts from the process automation domain stated that a major hurdle to availability testing is that realistic testing environments are often not available, as these can get extremely expensive. Furthermore, tests can hardly simulate long-time effects as they are typically only run over a short period of time.

Operation

In the interviews, the experts pointed out that availability is not alone influenced during the development of the system but also during the operation of the system. The two main topics that the experts deemed most relevant here are maintenance and monitoring.

Performing Maintenance (T21). Several experts said that in their respective company regular maintenance activities are carried out. This does mainly relate to computing hardware. During these maintenance activities certain parts that are subject to wear-out, are replaced (even if they are not yet broken). Sometimes, these maintenance cycles are also regulated by law. An example for such a law is the German law for the development and operation of tramways (BOStrab).

Monitoring the System (T22). In many cases, systems are continuously monitored. From the interviews we get two main goals that are connected to monitoring. The first goal is to assess the actual availability of the system. The second goal is to detect failures quickly and thus be able to react to them in a timely manner.

As an example of the first goal, one expert stated that they are regularly monitoring their system (a business information system) and calculate its availability. The monitoring results are included into monthly reports in the form of availability metrics. A different expert from the automation domain told us that they are monitoring a system during one week of operation recording all failures. From this they calculate a value for availability. Regarding the second goal, an expert from the medical domain described a system that possesses a dedicated monitoring processor.

When we asked for the concrete implementation of monitoring, we got rather diverse answers. In many cases the monitoring is performed on a rather technical level without relation to the actual functionality of a system. For instance, the processor workload, the activity of certain processes or the network traffic is analyzed. One expert told us that they are currently looking into pattern detection in these pieces of information. A common tool that was named by several experts and which is used for this purpose is Nagios[1]. As the information gathered with this kind of monitoring is rather coarse, several experts pointed out the problem that inferring, whether the system is currently operating or not, is not always possible from this data.

In two cases, we encountered a monitoring strategy on a more fine-grained level. One was the aforementioned case of a surveillance processor. In this case, the actual outputs of the main processor are continuously monitored and compared to the target outputs. Although this approach allows detailed monitoring, according to the corresponding expert, it comes with considerable effort as the monitoring routines are usually not trivial in order to achieve a good trade-off between strictness and tolerance. For instance, sometimes output values have to be buffered for a short period in order to allow for small delays between the two processors.

The third monitoring strategy was reported by an expert from the business information systems domain. In this case, the availability of the server part of a client-server application is being monitored. Therefore several test sequences are executed against the server in cyclic intervals. Usually, these tests do not cover the whole functionality, but instead only basic use cases (for instance logging in). They might even only trigger a certain diagnostic function. The application is considered available as long as the tests succeed. As soon as one test fails, it is considered unavailable.

[1]http://www.nagios.org

An additional aspect to monitoring was brought up by one expert. He emphasized that monitoring should be done in a proactive fashion. This means, monitoring should be done in a way that not only failures are detected but also conditions that will probably lead to an outage shortly.

Generally, monitoring is considered an important issue by the experts. Several experts said they are either currently planning to expand monitoring or that they would like to expand monitoring if there would be budget. The experts also stated that a major hurdle to monitoring is that their respective application does not provide suitable interfaces for the monitoring. Therefore establishing monitoring is cumbersome.

Documentation

Creating a RAM Case (T23). We asked the experts how decisions and analysis results regarding availability are being documented. In two cases (rail domain and automation domain) the experts reported that for each system, a RAM case is usually created. This document contains the availability goals and a detailed argumentation why and how these goals are met by the developed system. It contains, for instance, relevant architectural decisions with their rationales, models and results of FMEA, fault-tree or RBD analyses as well as the data on which the analysis was based upon (e.g. MTBF values of hardware components together with their source).

Creating a Manual (T24). In some cases the experts explained that within their organization, architectural decisions are usually documented in a manual. If availability is important for a system, such a manual may include descriptions how the architecture is supposed to guarantee a high availability. This form of documentation is less formal and less detailed than a RAM case.

Documenting Best Practices (T25). Apart from these system specific types of documentation, we also encountered the documentation of best-practices. These take, for example, the form of architecture patterns and anti-patterns the company has already made experience with. The information is either gathered in some central document or, as reported by one expert, is distributed individually, e.g. via E-Mail.

2.3.5 RQ 4: Problems

All experts claimed that major availability problems due to design flaws are rare at their respective companies. Reasons for this are advanced infrastructure technologies, such as virtualization, and the gathered experience in their organizations. However, during the interviews the experts still reported on several availability related problems, they perceive. The problems can be categorized into problems regarding specification and analysis of availability on the one hand and specific threats to availability on the other hand.

Problems Regarding Specification and Analysis

Formulating Availability Requirements (T26). We saw that high-level availability requirements are often motivated by economic considerations. These high-level requirements often come in the form of metrics. Two experts from the business information system domain stated that it is hard to find an operationalized notion of these availability metrics for a concrete system. One of these experts explained that the understandings of availability of the business departments and the responsible technical departments do not always match and need to be discussed. In this discussion, the business departments can get overwhelmed, according to the expert. Therefore, instead of using availability metrics, qualitative requirements for the system's failure behavior are developed together with the business experts.

Understanding System Dependencies (T27). In order to evaluate an architecture with regard to availability, it is crucial to understand the dependencies between elements of the system. With dependencies the experts related to cases where some element's behavior is influenced by the outputs or the state of a different element. An expert from the automation domain explicitly stated that the biggest problem, when performing availability analyses, is that an engineer overlooks some of the dependencies that are present in the system. Undiscovered dependencies can be the cause for availability problems during the operation of a system. An expert from the train protection domain described an example where the switch-over time of a redundant network component exceeded a certain predefined threshold of a safety monitoring routine. This routine initiated the shutdown of a great part of the system as the switch-over happened. In this case, the subtle interaction between a fail-over routine and a safety functionality resulted in an availability problem. Finally, even if dependencies are uncovered, assessing their impact quantitatively is regarded difficult by the experts.

Effort for Analysis and Verification (T28). A problem, reported by an expert from the automation domain, is the high effort needed to analyze a system's availability. This problem is especially prominent in a setting, where a rigorous analysis is performed by availability specialists. These specialists possess knowledge on analysis methods but have to familiarize themselves with the system at hand. The expert noted that usually not all relevant information, especially regarding dependencies, can be drawn from design documents. Hence, the specialist needs to work closely together with the engineers of the system, for instance in the form of common workshops. We saw that a different method for verifying availability is testing. However, the main problem here is the effort needed to create realistic testing environments. Furthermore, availability testing comes with further effort as short testing periods have to be compensated by deliberately introducing faults into the system.

Granularity and Reuse of Availability Analyses (T29). An expert from the rail automation domain formulated a problem that is related to the type of systems that they deal with. In this domain, it is common that there exists a generic system, which is then adapted for the specific context of a customer. However, it is hard to analyze the availability of the final system, deployed at the customer, on the basis of the generic system. In the past, the availability analysis hence needed to be re-done. According to the expert from the rail domain it would be helpful if the availability analysis on the generic system was modular enough and on suitable level of granularity such that partial results can be re-used for subsequent analyses of the deployed system.

Monitoring (T30). An important point that was stressed by several experts was the importance of monitoring during system operation, both for assessing the availability and for being able to react quickly to disturbances. Often, the information gathered during monitoring is rather coarse. Several experts pointed out the problem that inferring, if the system is currently operating or not, is not always possible from the gathered data. However, depending on the level of detail, building monitoring facilities can be expensive. Some experts hence stated that more and more detailed monitoring would be helpful, but there is often no budget.

Specific Threats to Availability

Partial failures (T31). A problem, described by an expert from the business information system domain, relates to the way components of system fail. According to the expert, components that either fail completely or are fully functional are usually not a threat to system availability, as failures can be efficiently detected and either compensated or repaired. However, components that fail partially pose a problem, as the failures may affect availability but are harder to detect and repair.

Spare Parts (T32). Especially for long living systems, it is important that spare parts can be obtained in future. Two experts in our interviews pointed out that for some of their systems, the acquisition of spare parts is difficult, as the system was built several decades ago and there have been several changes in technology. Thus, the use of outdated technology can pose a threat to availability, as repair times may get longer. However, the use of outdated technology may sometimes be due to regulations. For example, one expert reported that for one of their systems, they have to use monitors with the aspect ratio 4:3, which are increasingly rare today.

2.4 Discussion

The interview results show that the availability of software-intensive systems is a relevant topic in the industry. For two of the experts we interviewed, analyzing a system's availability and consulting projects with respect to availability is even their main task. In several domains, for example the automation, train protection and (with less emphasis) the medical and business information systems domains, availability properties are explictly demanded by customers and/or regulators and need to be proven (T1,T2). Even in cases where availability requirements were not quantitatively specified and a rigorous proof is not required, we saw that nevertheless considerable effort is spent to ensure availability and to monitor systems accordingly (T3). However, the availability of a system is always subject to economic considerations (T4).

Although the experts we interviewed share the same intuitive meaning of availability as a measure of the time the system is operating, there are also differences in understanding. More precisely, we observed different notions of failure, regarding the rigor of failure definitions (T5), the perspective on the system (T6,T7), and the classification of failures (T8). We further found that availability is either grasped quantitatively via a range of metrics or qualitatively, for instance by properties of the system's behavior (T9).

There are many activities that contribute to reach availability goals. These activities relate to different phases of the product's lifecycle including project management (T10), requirements engineering (T11,T12), architecture development (T14-T18), software development (T19,T20) operation (T21,T22), and documentation (T23-T25). These strategies are not only related to technical issues. Many experts stressed the fact that other aspects such as planning for maintenance (T21) are also important. Still, the two topics that receive most attention are system architecture and monitoring. The system architecture, is perceived as the most important lever for building highly available systems. Most techniques that are employed within the architecture are based on some form of redundancy (T14). However, other design paradigms, such as "design for failure" (T16), also receive attention.

With the help of advanced technology such as virtualization (T14) as well as experience, the companies generally succeed in building highly available system. Nevertheless the experts named a number of problems. One kind of problem is related to the specification and analysis of the system availability. The analysis is cumbersome, as availability is often hard to operationalize and therefore formulating requirements is difficult (T26). The effort to understand the system and its dependencies (T27) as well as the efforts for applying analysis methods is high (T28), and modularization of the analysis is difficult (T29). The experts also identified some system characteristics that in their experience often pose a threat to availability (T31,T32).

2.5 Threats to Validity

In this section we address possible threats to the validity of our study. We structure the threats into descriptive validity, interpretive validity and external validity as proposed for qualitative research, e.g. by Johnson (1997). We leave out internal validity as in this study we do not suggest any cause-effect relationships.

2.5.1 Descriptive Validity

Descriptive validity relates to the problem that we did not correctly protocol the interviews or that our protocols are ambiguous and do not faithfully represent what the interviewee said. As we did not record the interviews, our result are in principal affected by this threat. However, we performed the interviews with two persons, one concentrating specifically on minute taking. Furthermore, the minute-taking person was instructed to write the protocol as literally as possible. Additionally, the protocols were reviewed by the interviewing person right after the interview. Finally, a first draft of this study report was presented to the interviewees in request for feedback. We only received feedback from three of the experts. However, those that did respond acknowledged that the report is accurate. Therefore, we hope to minimize distorting influences due to the lack of recordings.

2.5.2 Interpretive Validity

Interpretive validity concerns the validity of how we interpreted the statements of the interviewees. In order to gain confidence that our interpretations are valid we presented the report to the interviewees, as mentioned above. This feedback did not indicate any wrong interpretations.

2.5.3 External Validity

External validity relates to the generalizability of the results. We do not claim that our results can be generalized, as with our study setting, a generalization is not possible. However, we tried to select our experts in a way that we include different domains, different types of systems, different companies and different expert roles. However, we still could only cover a small section of the field. For example, all companies that we included in our research were based in Germany, which could mean our results are biased in this respect.

2.6 Summary

In this chapter we reported on a qualitative interview study on the topic of availability of software intensive systems in the industrial practice. The interviews were performed with 15 experts in eight different companies from different engineering domains. The goal of the study was to describe the state-of-the-practice regarding availability. We described the results of the interviews along four main research questions: relevance of availability, understanding of availability, strategies to achieve availability and problems in the current practice. Finally we summarized and discussed our findings, as well as possible threats to the validity of this study.

The key finding of our study are:

- Availability is of different relevance for different types of companies and in different domains, depending, for example, on industry standards and regulations, the type of customers or economic considerations.
- The experts used different notions of failure and availability depending on company and context. Many experts used a quantitative notion of availability that refers to a black-box view on the system.
- Availability related activities are carried out in all phases of the development lifecycle from requirements engineering until system operation and maintenance.
- The experts report on several difficulties regarding requirements formulation and design-time availability analysis. Moreover, availability considerations pose inherent challenges, such as obtaining needed spare parts over a long period of time.

Chapter 3

State of the Art

In this section, we review the state of the art regarding the specification and analysis of availability properties. More specifically, we discuss work on modeling availability requirements, on model-based design of highly available systems and on models for availability analysis. In each section, we present the current state of the art as well as open issues connected with different streams of research and relate the presented work to our thesis.

Section 3.1 outlines research regarding the model-based elicitation, documentation and analysis of availability requirements. We discuss generic frameworks for non-functional requirements as well as approaches specific to availability or the broader field of dependability.

Section 3.2 summarizes work concerning the model-based design of highly available systems. This work includes comprehensive modeling languages and methods as well as pattern-based approaches. Although our thesis does not propose a design method we argue that these approaches could be integrated with our approach.

Section 3.3 reviews research that proposes models for the analysis of a system's availability properties. We first consider classical combinatorial models such as fault-trees and RBDs as well as their extensions. Afterwards we investigate different types of approaches that include the system architecture into the analysis models. Another stream of research considers availability from a user perspective. Finally, we consider generative approaches for the creation of availability analysis models.

3.1 Modeling Availability Requirements

Several authors address the structured and model-based elicitation and specification of availability requirements. Most authors, however, take a broader point of view and focus not only on availability but on dependability, which also includes related aspects such as safety, security and reliability. An even broader perspective is taken by authors that consider all types of quality requirements. We first review models for these general quality requirements in Section 3.1.1. Nevertheless, there are also specific models for availability and dependability requirements, which we discuss afterwards in Section 3.1.2. Finally, we summarize a specific stream of research that captures dependability requirements by analyzing behavior deviations in Section 3.1.3. Our thesis contributes to this field by providing a set of modeling artifacts to capture availability requirements in Chapter 5 and a method to create these artifacts in Chapter 6.

3.1.1 General Models for Quality Requirements

Availability is often termed a quality requirement[1]. *Quality models* aim to structure and operationalize quality properties and thus aid the elicitation and documentation of quality requirements. The quality model outlined in the industrial standard ISO/IEC 25010 (ISO/IEC, 2011) structures product qualities hierarchically and lists availability as sub-characteristic of reliability. However, the ISO model does not provide details on how availability should be measured for a concrete system. There are other quality models that aim to be more specific in how measures are attached to the quality characteristics (e.g., Deissenboeck et al., 2007; Kitchenham et al., 1997; Lochmann, 2014). By instantiating such a quality model and defining concrete measures, it is possible to formulate concrete and verifiable requirements on these measures (or aggregates thereof). However, the quality models provide no guidance how to specify measures and none of these quality models has so far been instantiated for availability.

A different type of model with a similar target, are goal oriented frameworks. The core idea of such a framework is to provide a modeling language for formulating high-level goals, specifying their relationships and refining goals to sub goals and to concrete requirements. Often, goals are captured in And/Or graphs extended by special annotations. Examples for such frameworks are the AGORA method due to Kaiya et al. (2002) or the NFR framework due to Chung et al. (Chung et al., 2012; Cysneiros and do Prado Leite, 2004). The strength of these frameworks are their ability to structure and relate high-level goals of different stakeholders. However, similar as for the quality models discussed before, they provide no further help in defining measures for the goals and requirements. Furthermore, they do not explicitly support availability requirements.

Relation to our work. A goal of this thesis is to provide a means for formulating precise availability requirements. The work described above pursues a similar goal for a wider range of requirements. The general frameworks for non-functional requirements aim to provide a language that can be used while eliciting and documenting such requirements and relate them to other types of requirements or activities in the system's lifecycle. However, the models and methods mentioned above are rather generic and do not capture the specific aspects of availability. They resort to natural language or externaly specified availability metrics to state availability goals or requirements. They do not provide any modeling means and guidance to define concrete availability measures. Although they often have a formal syntax to structure quality factors or goals, there is no formal semantics in terms of the behavior of the system. This, however, is what we contribute in this thesis. In Chapter 5, we suggest models for capturing availability requirements using a formal modelling theory and in Chapter 6, we provide guidance on how to develop such models for a concrete system. We see our work not as a rivalling approach to quality models but as a supplement, carrying on the requirements specification where the quality models stop.

3.1.2 Specific Models for Dependability Requirements

The literature often treats availability as part of the wider aspect dependability (Avižienis et al., 2004). Hence, apart from generic quality models reviewed in the previous section, we review

[1]Another term in this context is *non-functional requirement*, in contrast to functional requirements. This naming, as well as the distinction betweeen functional and nun-function has, however, often been critized (Broy, 2015; Glinz, 2007).

specific models for dependability requirements. As availability is one aspect of dependability in this context, these models are meant to capture availability, as well. Similar to the quality models, these models structure the concepts relevant for dependability aspects.

Rossebeø et al. (2006) argue that availability requirements depend on the concrete functionality delivered by a system as well as its possible functional degradations. They should therefore be expressed as properties of the system's behavior. Rossebeø et al. decompose availability properties into accessibility and exclusivity, which is due to their security point of view. An example for an accessibility type of property would be: "*The probability that an authorized user is denied access to the service at a given time t should be less than x*". Their conceptual model also includes threats and means for mitigation into account. See Figure 3.1 for an overview over the Rossebeø et al. model. Although this model is created explicitly for availability and includes a refinement of availability into accessibility and exclusivity it still does not provide further help for the specification of concrete accessibility and exclusivity properties.

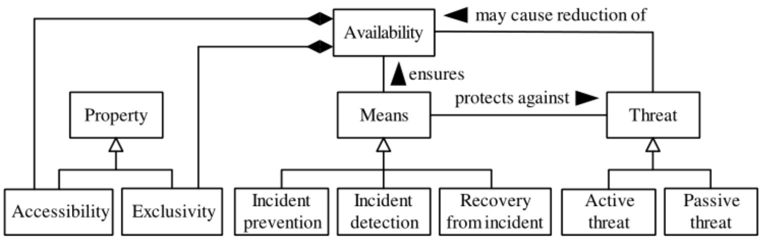
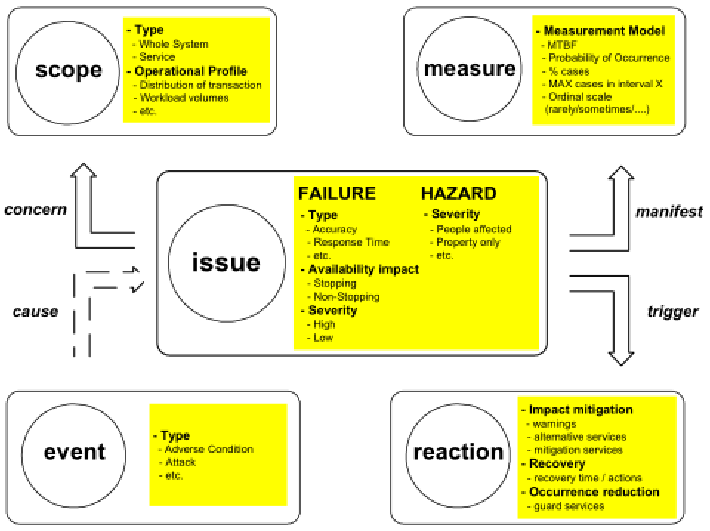


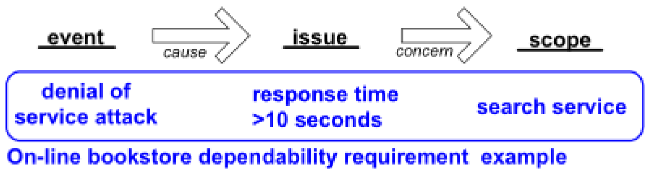
Figure 3.1: Overview over the conceptual model of service availability due to Rossebeø et al.. Illustration taken from Rossebeø et al. (2006).

Basili, Donzelli and co-workers (Basili et al., 2004; Donzelli and Basili, 2006) propose the "Unified Model of Dependability" (UMD) as means to elicit and specify dependability requirements, among them availability. The model incorporates a number of description concepts to capture different kinds of dependability properties. The three core concepts are *issue*, *scope* and *event*. An issue here relates to a certain failure or hazard which may be caused by an event, which is a potentially harmful condition of the system or a harmful action performed on the system. The scope relates to the part of the system that is considered, for example a specific system function or component. A scope definition may also include the operational profile connected with the given part of the system. In order to quantify dependability requirements, the concept *measure* is introduced and finally the concept *reaction* allows to reason about mitigation strategies. These core concepts can be refined for a specific dependability attribute and for a specific system in order to capture the stakeholders' requirements. See Figure 3.2 for an overview of these concepts and a small example. UMD is intended to be used by the different stakeholders by iteratively specifying actual issues that relate to a certain scope together with possible events that could cause the issue. By repeating and refining this specification (e.g. operationalizing the specification by giving quantitative measures), a dependability specification is reached.

Bernardi and her co-workers (Bernardi et al., 2011a, 2010, 2011b) developed an approach for dependability requirements that is closely coupled to the Unified Modeling Language (UML) and the UML profile MARTE (OMG, 2009). Their approach consists of a domain model for dependability (DAM) that structures, describes and interrelates dependability related concepts, such as dependability metrics (e.g. steady-state availability), threats to a system (including



(a) Concepts in UMD


event
cause
issue
concern
scope
denial of service attack
response time >10 seconds
search service
On-line bookstore dependability requirement example

(b) Example of UMD instantiation in case of an online bookstore

Figure 3.2: The main concepts used to capture dependability requirements in the UMD model described by Donzelli and Basili. Illustrations taken from (Donzelli and Basili, 2006).

faults) or fault-tolerance mechanisms. Figure 3.3 shows three submodels of the DAM domain model. The core submodel is shown at the top of Figure 3.3. It exhibits the basic concepts *Component* and *Service* with attributes such as *ssAvail* denoting steady-state availability. The *Threat* submodel is shown on the left. It contains at its core the concepts fault, error and failure interconnected by cause and effect relations. Each of them can be described with attributes such as *occurenceRate*. Failure descriptions can be assigned to services while faults can be assigned to components. In the *Maintenance* submodel, maintenance actions such as repair, recovery or replacement with spares can be specified and assigned to services or components. The elements from the DAM domain model can be used to annotate elements in a UML diagram. In a use case overview diagram, services are represented as use-cases and can be annotated with dependability requirements for these services. On the other hand, threats are represented as misuse-cases and are annotated with information on fault generation. For a detailed description of use-cases and misuse-cases further UML diagrams (such as sequence diagrams) are used, which can again be annotated using the DAM domain model. Also non-UML models such as fault-trees are integrated into the use-case model, e.g. to describe the misuse-cases. Bernardi et al. embed the modeling approach into the Rational Unified Process (RUP) (Bernardi et al., 2010) and give a step-by-step guide how different modeling activities are integrated into the requirements related phases of the RUP process in order to elicit and document reliability and availability requirements. Apart from the works of Bernardi, there are several other authors that

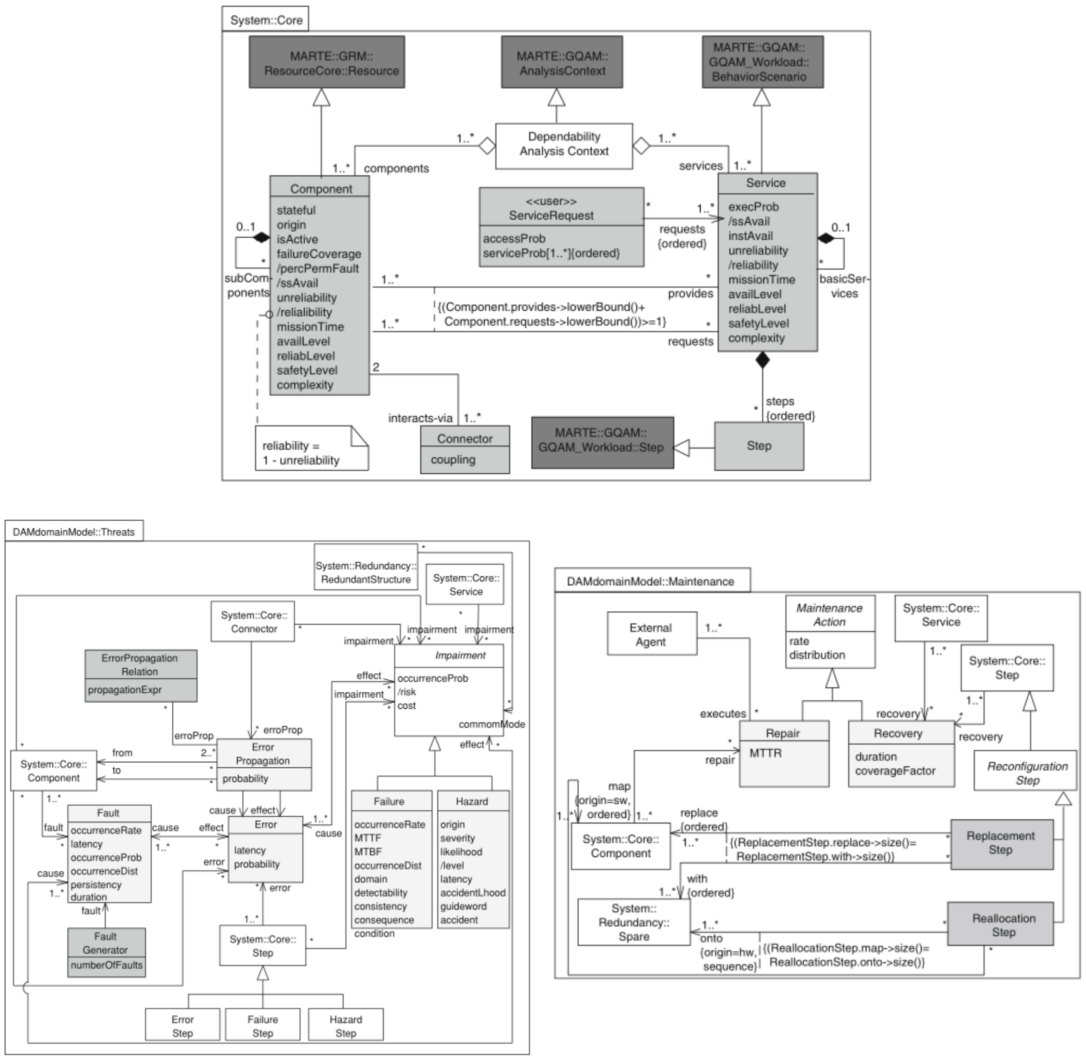


Figure 3.3: The DAM domain model for dependability including the core submodel (top), a submodel for threats (left) and a submodel for maintenance and fault-tolerance (right). Illustrations taken from (Bernardi et al., 2011b).

attempt to leverage the UML as a basis for eliciting and specifying dependability requirements, such as Allenby and Kelly (2001) and Johannessen et al. (2001), who both employ a guide word technique to identify possible failures based on use-cases as a basis for requirements elicitation. A systematic review on this literature can be found in the publication by Bernardi et al. (2012).

Relation to our work. Availability or dependability specific methods for eliciting and documenting requirements refine the generic methods discussed in the previous section as they aim to provide a suitable vocabulary specific for availability. By providing a structured concept model of availability they support eliciting and documenting availability requirements. While especially the model by Rossebeø et al. stays vague in how availability specifications should look like, the model by Bernardi et al. gives more guidance. However, all models miss a semantic ground on which they can be interpreted. The specifications are informal or at least semi-formal. The actual interpretation of the effect of the annotations on the system's behavior is left unspecified. In Chapter 5 of this thesis, we propose models based on a mathematical modelling theory. Therefore, we obtain a precise and rigorous interpretation of availability requirements in terms of the system's behavior.

Especially the model by Bernardi et al. aims at integrating the availability related concepts into the actual system models. Using the DAM profile, it is possible to add annotations to use case models as well as to architecture models. In our work, we similarly relate availability requirements models to other (requirements) models of the system in order to ensure that the requirements are meaningful and to enable an automated verification.

In one aspect the discussed work exceed the scope of this thesis. All models provide explicit means to model mitigation techniques or fault-tolerance techniques on a high level of abstraction. While in our approach, these aspects can be captured by the underlying modeling framework, we miss the high-level abstractions for these aspects.

3.1.3 Requirements through Behavior Deviation Analysis

A different line of research proposes to systematically analyze possible deviations of a system's behavior to elicit dependability requirements. This research is in the tradition of safety analyses employed in mechanical or process engineering. It adopts methods from traditional safety analysis and adapts them to software intensive systems. An example is the SHARD method (McDermid and Pumfrey, 1994; Pumfrey, 1999). The SHARD method involves a step-wise analysis of the system departing from a data-flow model of the system. One of the core steps is to study possible behavior deviations with the help of guide words. The guide words proposed by Pumfrey are *omission*, *commission*, *early*, *late* and *value*. After possible deviations are identified, the SHARD method stipulates to investigate possible effects of the deviation and develop mitigation proposals. These proposals are the requirements that result from the application of the method. Despotou and Kelly (2006) discuss, how such a method can be adapted for dependability. In both of the above approaches, guide words denoting classes of behavior deviations are used to identify possible failure modes of the services that a system is supposed to deliver.

Relation to our work. The line of research that that we reviewed above captures dependability requirements through deviating behavior. This has the advantage that the requirements possess a clear link to the system's behavior. We adopt this approach as part of our method. In Chapter 6, we suggest to define failures by specifying possible deviations from the nominal behavior. In the same chapter we also take up the core idea of the works by Pumfrey, McDermid, Despotou and Kelly to analyze possible behavior deviations with the help of a set of guide words based on the original guide words suggested by Pumfrey. A difference between the discussed approaches and our work is that their primary goal is to elicit requirements in the form of mitigation strategies (i.e. solutions) whereas we are interested in formulating availability requirements that do not restrict the solution space. Furthermore, the approaches are not embedded into a model-based engineering methodology. Neither are they grounded on a formal modeling theory which hinders automated analyses. With our work, we aim to use the advantages of these deviation oriented requirements, but base them on a formal modeling theory and integrate them into a comprehensive engineering method.

3.2 Model-based Design of Highly Available Systems

A different stream of research is concerned with model-based methods to support the design of high-availability systems. We will first outline work that proposes a comprehensive design method for fault-tolerant systems in Section 3.2.1. Afterwards, we discuss approaches that aim to optimize system architectures with respect to dependability attributes in Section 3.2.2. Such approaches usually run under the label design space exploration. Finally, we summarize work that employs architecture patterns to support the design of highly available systems in Section 3.2.3.

3.2.1 Model-based Design Methods

In his Ph.D. thesis, Buckl (2008) introduces FTOS, a model-based approach to design fault-tolerant systems. In its core, FTOS provides a modeling language to capture the input/output behavior of a system and provides additional abstractions to specify assumptions on faults (such as fault-rates and fault-effects) as well as fault-tolerance mechanisms (such as redundancy). Based on these models Buckl describes verification techniques and develops code generation facilities.

Relation to our work. Providing a design methodology for high-availability systems is not the main focus of this thesis. However, by building on a formal modeling theory as well as a method for seamless model-based development, we have to some degree a design methodology built into our approach. Furthermore, we adopt techniques for modeling faults (cf. Section 6.4.3). In a case study of a train door control system (Chapter 7), we employed the underlying modeling framework to model a fault-tolerance mechanism. Nevertheless, our main aim is to provide means to formally model availability requirements such that they relate to the functional requirements of the system and that they can be easily verified against a given design.

3.2.2 Design Space Exploration

While in the FTOS approach by Buckl, the design is created manually, there are approaches that aim to automate this task. Under the term design space exploration (DSE), several authors (e.g. Bolchini et al., 2001; Janakiraman et al., 2004; Jhumka et al., 2005; Streichert et al., 2007; Xie et al., 2004) investigate ways to search for architecture configurations that provide high availability. Grunske et al. (2007) provide an overview over research in this direction. They propose an abstract method for design space exploration with respect to dependability and discuss open problems in this field. According to Grunske et al., the basic elements of a design space exploration method with respect to dependability are the following:

Dependability Evaluation/Prediction Based on Architecture Specifications: These are methods that take an architecture model as input and deliver a quantitative evaluation of a dependability attribute. Grunske differentiates between analytical and simulation based methods.

Dependability Improving Measures: These measures are transformations that can be applied to the architecture in order to improve the system's dependability. A typical measure is to introduce redundancies for certain elements in the system. Starting from an initial architecture the application of the transformations spans the design space.

Optimization Strategy/Design Space Exploration: This relates to the combinatorial problem to search the design space for the optimal architecture with respect to one or several dependability attributes.

The design space exploration process proceeds from a given architecture by first evaluating the architecture with respect to dependability. From this outcome a decision is made to look for architecture optimizations. The current architecture candidate is then searched for possible improvements. Some of the possible improvements are selected based on the global optimization strategy and the process starts a new iteration.

Relation to our work. Our main objective is not to provide a design method for highly available systems. However, referring to the elements of a DSE method as outlined by Grunske and Zhang, we contribute to the field by providing a technique for the automated availability evaluation of an architecture. In Chapter 7 we present a prototypical toolchain for availability modeling and assessment. Such a tooling could be used as part of a generic DSE tooling for availability.

3.2.3 Architecture Patterns for Availability

There are attempts to ease the construction of highly available systems by providing architectural patterns for availability. Examples for such an approach can be found in work on architecture by Bass et al. (2013), extended by Scott and Kazman (2009). Another example is the work by Saridakis (2002). In all these works, a range of architectural *patterns* (or more small-scale *tactics*) are presented that can be employed in concrete architecture models to achieve a high availability. Most of these patterns relate to fault-tolerance or fault-detection. An example is the *watchdog*-pattern that prescribes a timer that needs to be regularly reset by the monitored component. If this does not happen, an alarm is raised which can be used to switch to a redundant component.

Relation to our Work. Providing patterns for highly available architectures is not our main focus. An integration of a pattern-based approach for architecture improvement with our approach is nevertheless promising as the effect of the patterns could be analyzed with respect to the availability requirements. We do provide modeling patterns and basic modeling building blocks in Chapter 6, however not for the system architecture but instead for the definition of failures and for the definition of availability metrics.

3.3 Availability Analysis Models

A large body of work deals with models of systems created for the purpose of analyzing their availability properties. We start in Section 3.3.1 by reviewing classical combinatorial models, such as reliability block diagrams (RBD) and fault trees (FT) as well as their extensions. Afterwards we summarize work that aims to integrate these combinatorial model into high-level architecture models in Section 3.3.2. Further work has been done in the field of architecture-based availability prediction which we discuss in Section 3.3.3. We outline approaches that consider availability from the perspective of the user instead as from the perspective of the system in Section 3.3.4 and finally Section 3.3.5 picks up work that discards manually created analysis models in favor of automatically generated models.

3.3.1 Combinatorial Models

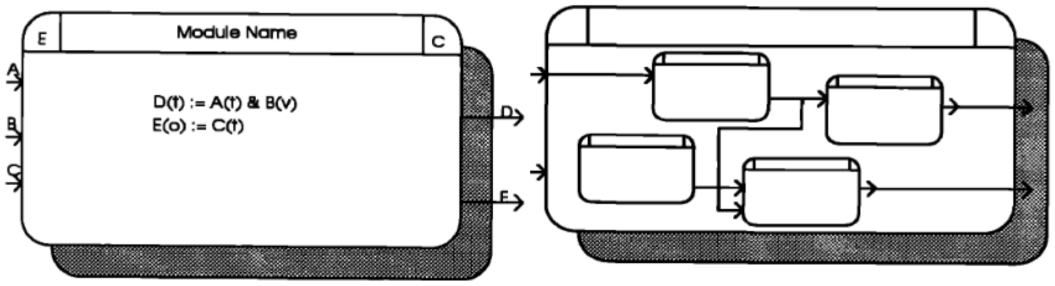
There are established modelling techniques such as Reliability Block Diagrams (RBD) (Birolini, 2010) and Fault Trees (IEC, 1990) that are commonly used to analyze the reliability and availability of technical systems. In both cases, the diagrams model which (combination of) elementary faults lead to a failure of the whole system or a system function. Even though the representation of RBDs is closer to a system's architecture, an RBD is nevertheless a diagram of system events rather than of system components (Birolini, 2010). A problem with these kinds of models is that only rather simple scenarios can be captured (Birolini, 2010). For example, sequence dependencies between element faults cannot adequately be modelled. To approach this problem, several extensions of the two formalisms have been proposed, such as Dynamic RBDs (Distefano and Puliafito, 2009) and Dynamic Fault Trees (Bechta Dugan et al., 1992). These extensions introduce new primitives into the modeling languages that allow to model more complex, possibly time-dependent dependencies between faults. A second problem is that fault tree models and RBDs do generally not match the architecture of the system that is modelled. As the basic elements in fault trees as well as RBDs are events (faults) and not architectural entities such as components or ports, a modularization of these models is often skewed compared to the system architecture (Domis and Trapp, 2009; Kaiser et al., 2003).

Relation to our work. Analyzing and predicting the availability of a system based on system models is one of the core concerns of this thesis. The combinatorial models, such as RBDs and fault trees, serve exactly this purpose. They are built in order to derive values for availability metrics from them. The main problems associated with these models are their restricted expressiveness. Complex availability requirements cannot be formulated with these models. Furthermore, it takes much effort to reliably gather the information about the effect of faults needed to create the models. Additionally, maintaining these models can cause a considerable overhead in case of large systems as they have to be constantly adapted to changes in the system. We avoid these weaknesses in the following way: In Chapter 5, we introduce an artifact model for availability, containing specific models for availability requirements. We show in the same chapter how to perform availability analyses based on the availability models, by leveraging the original system models and thus avoiding any redundancy.

3.3.2 Mixed combinatorial/Architecture Models

Research has been conducted to unify architecture models with combinatorial analysis models. Examples for early approaches are the Failure Propagation and Transformation Notation (FPTN) by Fenelon et al. (1994) as well as Component Fault Trees (CFT) due to Kaiser et al. (2003). In FPTN, a system is described by a set of modules connected by inputs and output channels. However, these channels represent failures that propagate from one module to another. A module may either contain hierarchically another module network, or a logical description how outgoing failure depend on incoming failures. An FPTN model can then be used as a basis for a numerical analysis. See Figure 3.4 for an example of an FPTN model.

The core idea of CFTs is to describe the fault behavior of components with component specific fault trees together with an interface describing by which faults of external components the component is influenced and which faults it produces. The overall fault tree is then built up from these encapsulated component-specific fault trees. See Figure 3.5 for an example of a CFT. These basic approaches have been extended. For example, FPTN has been extended with probabilities by Ge et al. (2009). Domis and Trapp (2009) add a further type of failure propagation to CFTs. Finally, Grunske (2006) shows how such kinds of approaches can be integrated with a general approach to component-based design.



(a) FPTN Module with a failure propagation description. (b) FPTN Module with a nested module network.

Figure 3.4: Examples of the FPTN notation. Illustrations taken from Fenelon et al. (1994).

Relation to our work. Our work and these techniques share the core idea to integrate system architecture models with models for availability analysis. Our approach is to describe the different models and their relationships as part of a comprehensive artifact model. Compared to the purely combinatorial models, the mixed combinatorial/architecture models improve the compositionality of the analysis models as they allow the composition of the analysis model from sub-models that relate to subsystems. However, these approaches still lack the expressiveness to capture complex availability requirements and still come with considerable manual effort to understand and model the effect of faults in the system.

3.3.3 Architecture-based Availability Prediction

The research field of architecture based availability/reliability prediction takes the idea of an integration between architecture models and availability analysis models even further. An overview over this research field is given in the publications by Goseva-Popstojanova et al. (2001), Goševa-Popstojanova and Trivedi (2001), Gokhale (2007) and Immonen and Niemelä

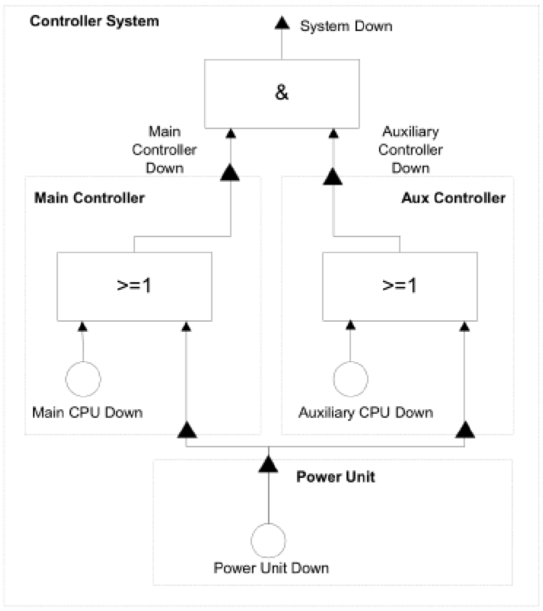


Figure 3.5: Example for a Component Fault Tree due to Kaiser et al. (Kaiser et al., 2003). The CFT for the *Controller System* contains the fault trees for the components *Main Controller*, *Aux Controller* and *Power Unit*.

(2008), however with a focus on pure software systems. In general, these approaches consider the architecture of a system in terms of components and connections between them and aim to deduce the availability or reliability of the systems given either the behavior as well as the failure behavior of the components or given information about the reliability/availability of the single components. Especially early articles assume a sequential control flow between the components, often modelled probabilistically. We illustrate this kind of models with the approach due to Littlewood (Littlewood, 1979), other examples are the work by Kubat (Kubat, 1989) and Laprie (Laprie, 1984). In the Littlewood model, an architecture is given by a set of components. The dynamic of the system is described by two structures. The first is a Markov chain that captures the transition of the control flow from one component to another and is given by the probabilities

$$p_{ij} = \mathbf{Pr}[\text{program control flow transits from module } i \text{ to module } j].$$

The second is a family of general probability distributions μ_{ij} where each describes the distribution of sojourn times for a component i when entered from component j. Additionally, the failure behavior of a component i is modelled by a Poisson process with parameter v_i. Based on this mathematical model, Littlewood derives an approximation for the overall failure process which can be used to obtain an approximation for the availability (Littlewood considers point- and interval-availability). In a classification by Goševa-Popstojanova and Trivedi (2001) these types of models are referred to as *state-based* and are contrasted to *path-based* models where the failure probabilities are not initially assigned to components but to whole execution paths covering multiple components. An example for such an approach is the work by Yacoub et al. (2004). In this work, the main input are scenario models that capture interaction sequences between the involved components. These scenarios are extended with execution probabilities.

From the scenarios Yacoub et al. construct a Component Dependency Graph (CDG), which captures the system components and probabilities for the transfer of control from one component to another. The CDG is then the base for the actual reliability or availability analysis. The last category in the classification of Goševa-Popstojanova and Trivedi are *additive models*, where the system availability is calculated from component availability values.

In the approaches discussed so far, an operational profile or environment model is only implicitly given by the transition probabilities of components or probabilities of scenarios. Furthermore, these approaches capture only either software or hardware faults but not a combination of both. Reussner et al. (2003) and later Brosch et al. (Brosch, 2012; Brosch et al., 2012) aim to overcome these weaknesses. As the work by Brosch is close to our work, we discuss it in greater detail: Brosch builds his work upon the Palladio Component Model (PCM) (Becker et al., 2009), a modeling language geared towards the analysis of reliability and other quantitative system properties. PCM supports the creation of the following modeling artifacts:

Architecture Model The architecture model comprises the software components of the system. Components expose connectors, termed required or provided roles in this context. Components can be composed by connecting provided and required roles appropriately.

Component Service Behavior Model The component service behavior model comprises abstract behavior descriptions for the roles present in the architecture model. The behavior is specified in terms of processes with the process actions referring to resources and calls to required services.

Deployment Model The deployment model represents hardware resources such as CPUs or hard disks and resource containers such as servers. The deployment model further describes the allocation of the components of the architecture model to the elements of the deployment model.

Usage Model The usage model describes different scenarios how the system is used at its interface by different types of users. A usage scenario is annotated with the probability that this specific scenario is executed. The scenario itself is modelled as a process where the actions are calls to the services provided by the system at its interface.

Brosch extends this basic models by annotations relevant for reliability prediction, such as fault probabilities of the hardware (e.g. MTTF and MTTR values) and fault probabilities of the software components. To illustrate his approach, Brosch uses the example of a simple library system. See Figure 3.6 for the example given by Brosch et al. (2012). In this example, there are two components, *SearchGuide* and *SearchEngine*, deployed on dedicated servers with CPU and hard disk resources. For the hardware resources, MTTF and MTTR values are specified. The service behavior model specifies that for a search, the *SearchEngine* uses either only the hard disk or both, hard disk and CPU resources, depending on which type of search is requested. The behavior of the *SearchGuide* is such that it either triggers the library search or the archive search depending on the year of the book that is searched. The usage model specifies a library visitor that searches a book of a certain year with a given probability.

The actual availability or reliability analyses are performed on this model by transforming the model in several steps into a Markov model and solving it. Regarding the analysis performance, Brosch reports that models with 20 resources can be solved within 1 hour.

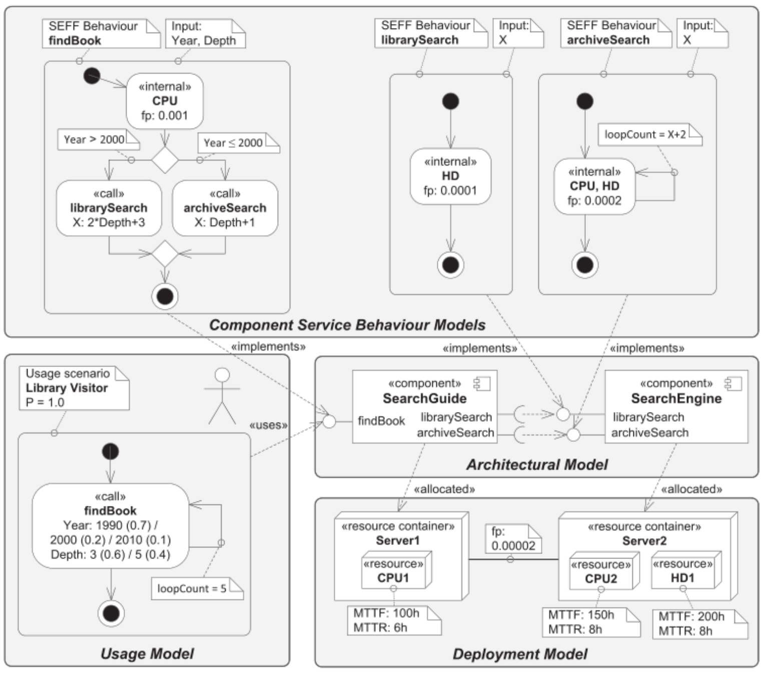


Figure 3.6: Example of a PCM model of a library service with reliability annotations. The example and the illustration is taken from Brosch et al. (2012).

Relation to our work. There is a close relationship between the work described in the previous section and our work. System models, extended with models of the failure behavior and/or system usage are employed for availability analysis. While earlier examples, such as the method due to Littlewood (1979) use rather simple system models, which do not fit to the kind of distributed systems that is pervasive today, more recent work use a family of interconnected models (such as logical architecture with behavior, deployment, usage, etc.). An example for the latter is the work by Brosch et al. (2012). In our work, we also build on and extend a comprehensive artifact model that reflects the system functionality, its logical and technical architecture and its environment (Chapter 5). Other than previous work, we strictly build on a view that availability relates to the behavior of the system. This enables us to further extend the state of the art by suggesting additional models to capture system specific notions of failure and availability that relate to the functional requirements of the system.

3.3.4 User-Perceived Availability

With the exception of Brosch, the approaches discussed so far consider availability as system availability. That means they derive the overall availability by solely considering the elementary components, neglecting how the users interact with the system. To see, that a user-centric view (or black-box view) may be different from a system-centric view (or glass-box view) think of the situation where a system has technically failed in a period where it is not used. Under the term user-perceived availability several authors aim to include this aspect in their models. In the work by Wang and Trivedi (2005), the analysis model consists of a system model and an explicit user model. The user model in this approach does not only serve as an operational profile but does also provide the reference for the used availability metric: “The service availability is the probability that all requests are successfully satisfied during the user session" (Wang and Trivedi, 2005). The user session in turn is defined by the user model. Wang and Trivedi represent the user model as a deterministic time Markov chain (DTMC) and the system model as a continuous time Markov chain and give an analytical expression for the service availability.

An approach to analyze the user-perceived availability of a workflow-oriented business system is presented by Milanovic and Milic (2011). The authors capture the user perspective on the system in a business process model (BPM). The activities in a BPM can recursively contain further processes. Ultimately, the atomic activities are mapped to basic services of the system. The services themselves are mapped to the infrastructure resources used to implement the services. These resources are annotated with MTTF and MTTR values. From this multi-level system model, Milanovic and Milic automatically generate analysis models such as RBDs or Markov models and use them for availability analysis. A similar, multi-level approach is pursued by Kaniche et al. (2003) for the example of a web-based travel agency.

Relation to our work. We share with this stream of research the core idea to consider availability from a user perspective instead of a system perspective. This means that only faults that actually lead to a deviation of the system's interface behavior that is outside of its specification and that matters to its environment should be considered from an availability point of view. The main difference between the work discussed above and our work is that previous approaches perform a translation (or a sequence of translations) of user visible failures to system-internal faults. The typical reasoning underlying these approaches is to deduce for a certain user operation a configuration of resources needed to support this operation. When this configuration breaks, the user operation is considered failed. The user behavior is modelled in terms of these operations and from the combination of user behavior and resource characteristics the availability can be calculated. Our approach is more consequent by defining availability only in terms of the visible system behavior and without explicitly translating the system-internal faults to failures. This enables us to formulate fine-grained definitions of failure and, thus, of availability requirements.

3.3.5 Automatic Generation of Analysis Models

The work by Milanovic and Milic (2011) we discussed in the last section is also an example for a class of approaches where the actual analysis models are derived from other models that are closer to the design models. A similar scheme is applied by Bernardi et al. (2010) which we already discussed in Section 3.1.2. Bernardi et al. propose to generate analysis models from the annotated UML diagrams (use cases and component diagrams). However, there is earlier work on the automatic synthesis of fault-trees from system models for safety analysis. See the publication by Lapp and Powers (1977) for an overview of early work and an approach that uses digraphs as a system model. More relevant for this thesis is work by Papadopoulos and Maruhn (2001) who use Matlab Simulink models as input and semi-automatically generate fault-trees from these models. To enable the fault tree generation, all components are subjected to a hazard analysis to determine their failure behavior. After this manual local analysis, a global analysis determines the failure propagation through the system and generates the fault trees. Majdara and Wakabayashi (2009) introduce a similar method to generate fault trees from a general component oriented description of systems. The approach is not only applicable to software-intensive systems but to general mechatronic systems, where the interaction between components can be described as flows (energy, material, information or commands). A component itself can be specified with input/output tables and state diagrams. The fault tree is generated by tracing back along the flow relationship and interpreting the behavior descriptions of the components in reverse direction to determine the conditions for a wrong output. A final example is the work by Joshi et al. (2007). In this case the Architecture Analysis & Design Language (AADL) (Society of Automotive Engineers, 2012) and its error model annex (Society of Automotive Engineers, 2006) are used to represent the system model. Departing from the system model, where components are annotated with error behavior using the modeling techniques from the error annex, a fault tree is generated that can be analyzed using external tools.

Relation to our work. Generating detailed analysis models from more high-level analysis models or system models has several advantages compared to the manual creation of analysis models. First, if the generation procedure is sound and complete, the consistency of the analysis model to the original model is guaranteed. Second, even though redundancies are introduced by the generation, these redundancies are not problematic as the changes are only applied to the original model and then the generation procedure is re-started. Third, no manual effort needs to be invested to create the analysis models. In our work, we also aim to achieve these advantages. However, we do not generate explicit analysis models but instead leverage directly the system models together with additional modeling artifacts, describing availability requirements, for the analysis. Therefore, we achieve the above advantages without generating new models.

Chapter 4

Background and Formal Foundation

In this chapter, we introduce the basic terms and concepts that will be used throughout this thesis. The chapter is structured into two main sections. In Section 4.1, we introduce a comprehensive, formal modeling method. This method is based on a mathematical theory of systems and encompasses an artifact model that allows the model-based description of various aspects of the system. In Section 4.2, we discuss availability, which is the central topic of this thesis.

4.1 Formal System Model

In this section, we introduce a formal theory to model software-intensive systems. We start by briefly establishing the mathematical background. Afterwards, we introduce a specific notion of system and present the system modeling theory Focus, which provides a formal underpinning for the study of software intensive systems. Subsequently, we review an artifact model that builds upon the system model and that provides a structure for a multi-view description of a system.

4.1.1 Mathematical Foundation

Logic

As background logic we use first-order logic throughout this thesis. A value in this logic is represented by a *term* t. A term is either a constant c, a variable v, or a function application $f(t')$. A formal statement is given by a *formula* φ. A formula is either an atomic proposition, for instance "$a \in A$", a conjunction $\varphi \wedge \psi$ of formulas, a disjunction $\varphi \vee \psi$ of formulas, a negation $\neg\varphi$ of a formula, a universal quantification $\forall x : \varphi$ over a variable x or an existential quantification $\exists x : \varphi$ over a variable x.

We write $\varphi[x_1, \ldots, x_n]$ to stress that the formula φ has the free variables x_k. We further write $\varphi[x_1 \mapsto t_1, \ldots, x_n \mapsto t_n]$ to denote the formula obtained from φ by substituting free variables x_k with terms t_k.

Numbers

We consider the natural numbers $\mathbb{N} = \{1, 2, 3, \ldots\}$ and the natural numbers including zero $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$. We further use the extended natural numbers $\overline{\mathbb{N}}_0 = \mathbb{N}_0 \cup \{\infty\}$, the real numbers

$\mathbb{R}$ and the extended real numbers $\overline{\mathbb{R}} = \mathbb{R} \cup \{+\infty, -\infty\}$. With $d_R(r,u) = |r-u|$ we denote the usual distance between two real numbers r and u.

Streams

Definition 1 (Finite Streams)**.** A *stream* of length n over the (countable) alphabet Σ is a function

$$\{0,\ldots,n-1\} \to \Sigma \; .$$

We write Σ^n for the set of streams of length n. With $\langle\rangle$ we denote the unique stream of length 0, also called the *empty stream*. Σ^* is the set of all finite streams, that means

$$\Sigma^* = \bigcup_{n \in \mathbb{N}_0} \Sigma^n \; .$$

Definition 2 (Infinite Streams)**.** An infinite stream over alphabet Σ is a function

$$\mathbb{N}_0 \to \Sigma \; .$$

Σ^∞ denotes the set of all infinite streams and $\Sigma^\omega = \Sigma^* \cup \Sigma^\infty$ the set of both, finite and infinite streams.

We employ the notation $x = \langle x_0, x_1, x_2, \ldots \rangle$ to describe streams in terms of their constituents, and $x.n = x_n$ to select the $n-$th element of a stream. With $|s|$, we denote the length of a stream, i.e. $|s| = n \Leftrightarrow s \in \Sigma^n$.

Definition 3 (Concatenation)**.** A finite stream x can be *concatenated* with another stream y, denoted by $x \cdot y$, for which the following holds:

$$\begin{aligned}
&\forall n \in \mathbb{N}_0, m \in \overline{\mathbb{N}}_0 : \\
&\quad x \in \Sigma^n \wedge y \in \Sigma^m \Rightarrow \\
&\quad (x \cdot y) \in \Sigma^{n+m} \wedge \forall i < n : (x \cdot y).i = x.i \wedge \forall i \le m : (x \cdot y).(n+i) = y.i \; .
\end{aligned}$$

Definition 4 (Prefix)**.** A stream x is called a *prefix* of a stream y if there is a stream z such that $x \cdot z = y$. In this case we write $x \sqsubset y$. If s is a stream of length at least n then $s{\downarrow_n}$ denotes the unique prefix of s of length n.

Given two streams x, y, we write $gcl(x,y)$ for the length of longest common prefix of x and y, given by

$$gcl(x,y) = \max \left\{ l \in \mathbb{N}_0 \; : \; l \le \min\{|x|,|y|\} \wedge x{\downarrow_l} = y{\downarrow_l} \right\} \; .$$

Note that, if $x = y$, then $gcl(x,y) = \infty$.

Definition 5 (Distance between streams)**.** In order to measure the distance of two streams we use the *cantor metric* d_C. For two streams x, y the cantor metric is defined by

$$d_C(x,y) = \begin{cases} 0 & \text{if} \quad x = y \\ 2^{-gcl(x,y)} & \text{otherwise} \end{cases}$$

Definition 6 (Filtering)**.** A further operation on streams is *filtering*. The filter operator

$$\cdot \circledS \cdot \; : \; \wp(\Sigma) \times \Sigma^\omega \to \Sigma^\omega$$

filters away messages from a stream. It is recursively defined by the following set of equations (Broy and Stølen, 2001).

$$\begin{array}{lcll} & & A \circledS \langle\rangle & = \; \langle\rangle \\ m \in A & \Rightarrow & A \circledS (\langle m \rangle \cdot s) & = \; \langle m \rangle \cdot (A \circledS s) \\ m \notin A & \Rightarrow & A \circledS (\langle m \rangle \cdot s) & = \; A \circledS s \\ rng(s) \cap A = \varnothing & \Rightarrow & A \circledS s & = \; \langle\rangle \end{array}$$

Probability Theory

We briefly introduce some basic concepts of probability theory. For details on these concepts, see Neubeck (2012) as well as introductionary texts on probability theory (e.g. Dudley, 2002).

Definition 7 (σ-field)**.** Given a set Ω, a set $\mathcal{F} \subseteq \wp(\Omega)$ is called a σ-field, if $\mathcal{F}$ is closed under complements and countable unions and contains $\varnothing$. Due to its closure properties, a σ-field also always contains Ω. The pair $(\Omega, \mathcal{F})$ is called a *measurable space* and the elements of $\mathcal{F}$ are called *measurable sets*.

If $\mathcal{G}$ is a subset of $\wp(\Omega)$, then with $\sigma(\mathcal{G})$ we denote the σ-field generated by $\mathcal{G}$, that is, the smallest σ-field that subsumes $\mathcal{G}$, which is guaranteed to exist. As detailed by Neubeck (2012), one can construct σ-fields $\mathcal{S}_n$ over finite streams of length n as well as a σ-field $\mathcal{S}$ over infinite streams. More precisely, $\mathcal{S}$ is the σ-field generated by the set of *basic cylinders*. A basic cylinder is a set of the form $\mathcal{C}(x) = \{y \in \Sigma^\infty : x \sqsubset y\}$, for $x \in \Sigma^*$. If a measurable space is extended with a function that assigns probabilities to measurable sets, we obtain a probability space.

Definition 8 (Probability Space)**.** A *probability space* is a triple $(\Omega, \mathcal{F}, \mu)$, where

- $(\Omega, \mathcal{F})$ is a measurable space,
- μ is a function $\mathcal{F} \to \mathbb{R}$ such that
  - $\mu(A) \geq 0$ for all $A \in \mathcal{F}$,
  - $\mu(\bigcup_{i=1}^\infty A_i) = \sum_{i=1}^\infty \mu(A_i)$ for pairwise disjoint sets $(A_k)_k$ and
  - $\mu(\Omega) = 1$.

A probability space for infinite streams $(\Sigma^\infty, \mathcal{S}, \mu)$ can be derived from consistent probability spaces $(\Sigma^n, \mathcal{S}_n, \mu_n)$ for finite streams. Then μ is the probability measure on infinite streams that is consistent with all probability measures μ_n for finite prefixes. For details of this construction, see Neubeck (2012). The key point is that a probability space on infinite streams is uniquely determined by fixing probability spaces for finite prefixes.

We will denote the set of possible probability measures for a set of streams Σ^∞ with $\mathbf{Pr}(\Sigma^\infty)$. We will furthermore use the convenient notation $\mathbf{Pr}[q(\mu)]$ as short for $\mu(\{\omega\} \, q(\omega))$ for a predicate on streams q.

Definition 9 (Measurable Function)**.** Given measurable spaces $(\Omega, \mathcal{F})$ and $(A, \mathcal{B})$. A function $X : \Omega \to A$ is called $(\mathcal{F}, \mathcal{B})$-*measurable* if, for every $B \in \mathcal{B}$, the preimage of B under X is contained in $\mathcal{F}$:

$$\forall B \in \mathcal{B} : X^{-1}(B) \in \mathcal{F} \ .$$

Definition 10 (Random variable)**.** Given a probability space $(\Omega, \mathcal{F}, \mu)$ and a measurable space $(A, \mathcal{B})$, a random variable is a function $X : \Omega \to A$ that is $(\mathcal{F}, \mathcal{B})$-*measurable*. A random variable induces a probability space $(A, \mathcal{B}, \mu_X)$ by

$$\mu_X(B) = \mu(X^{-1}(B)) \ .$$

In this context μ_X is called the *distribution* of X. In case A is at most countable, X is said to be *discrete*. The distribution of a discrete random variable may be specified by a *probability mass function*. This is a function $f : A \to [0, 1]$, such that

$$\sum_{a \in A} f(a) = 1 \ .$$

With $\mathbf{dist}(A)$ we will denote the set of probability mass functions for the set A. We will later typically be concerned with random variables of the form $\Sigma^\infty \to \Sigma^\infty$ as well as real valued random variables $\Sigma^\infty \to \overline{\mathbb{R}}$.

It is often interesting to determine the *expected value* $\mathbf{E}[X]$ of a random variable. Intuitively the expected value is the average outcome of a random variable if an experiment is performed many times. In measure theoretic terms, the expected value is the Lebesgue integral of the random variable with respect to the probability measure:

$$\mathbf{E}_\mu[X] = \int_\Omega X(\omega) \, d\mu(\omega) \ .$$

From standard measure theory (see e.g. Schilling, 2005) we cite the following theorem that enables us to derive a random variable as the pointwise limit of a sequence of random variables.

Theorem 1. *The point-wise limit (lower limit, upper limit) of a sequence of measurable functions is measurable. Formally, if $(X_i)_{i \in \mathbb{N}}$ is a sequence of measurable functions, then*

$$X(\omega) = \lim_{k \to \infty} X_k(\omega)$$

is measurable. The same holds for $\limsup$ *and* $\liminf$.

Metric Spaces

A metric space is a notion from topology. It is a set together with a distance function for the members of the set. Besides defining probability spaces on top of the set of streams, it is also possible to define metric spaces based on streams. By showing a particular relationship between the probability space and the metric space we can obtain an intuitive way to identify functions as measurable, which is the goal of this section.

Definition 11 (Metric space)**.** Given a set Ω, a *metric space* is a pair (Ω, d) where $d : \Omega \times \Omega \to \mathbb{R}$ is a distance function with the following properties. For all $x, y, z \in \Omega$,

1. $d(x, y) = 0 \Leftrightarrow x = y$,
2. $d(x, y) = d(y, x)$,

3. $d(x,z) \leq d(x,y) + d(y,z)$.

An example of a metric space is $(\mathbb{R}, d_R)$, the real numbers together with the absolute difference as distance function. But we can also define a metric space based on the set of streams using the Cantor metric.

Theorem 2. *The pair (Σ^∞, d_C) of the set of streams together with the cantor metric forms a metric space.*

Proof. We have to show that d_C fulfills the following conditions for any $x, y, z \in \Sigma^\infty$

1. $d_C(x,y) = 0 \Leftrightarrow x = y$,
2. $d_C(x,y) = d_C(y,x)$,
3. $d_C(x,z) \leq d_C(x,y) + d_C(y,z)$

Conditions 1 is obviously fulfilled by the definition of d_C. Condition 2 is fulfilled, as gcl is symmetric in its arguments. We now show condition 3. If some of x, y, z are equal then condition 3 is true, as then the left part of the inequality also appears in the sum on the right side. As d_C is always greater or equal to 0, the sum on the right cannot be smaller than the term on the left. We now assume x, y, z are pairwise distinct. We consider two cases:

Case 1 ($gcl(x,z) \leq gcl(x,y)$)**.** If the longest prefix that x shares with z is smaller or equal to the longest prefix x shares with y, then z shares with y the same prefix it shares with x. Hence $gcl(y,z) = gcl(x,z)$ and thus

$$\begin{aligned}
& d_C(x,z) \leq d_C(x,y) + d_C(y,z) \\
\Leftrightarrow\ & 2^{-gcl(x,z)} \leq 2^{-gcl(x,y)} + 2^{-gcl(x,z)} \\
\Leftrightarrow\ & 0 \leq 2^{-gcl(x,y)}
\end{aligned}$$

Case 2 ($gcl(x,z) > gcl(x,y)$)**.** In this case, z shares with y the same prefix as x shares with y. Therefore $gcl(y,z) = gcl(x,y)$. It follows

$$\begin{aligned}
& d_C(x,z) \leq d_C(x,y) + d_C(y,z) \\
\Leftrightarrow\ & 2^{-gcl(x,z)} \leq 2^{-gcl(x,y)} + 2^{-gcl(x,y)} \\
\Leftrightarrow\ & 2^{-gcl(x,z)} \leq 2 \cdot 2^{-gcl(x,y)} \\
\Leftrightarrow\ & 2^{-gcl(x,z)} \leq 2^{-gcl(x,y)-1} \\
\Leftrightarrow\ & 2^{-gcl(x,z)} \leq 2^{-(gcl(x,y)+1)}
\end{aligned}$$

As $gcl(x,z) > gcl(x,y)$ there is a $k \geq 1$, such that $gcl(x,z) = gcl(x,y) + k$. Therefore, we can write the above inequality as

$$2^{-(gcl(x,y)+k)} \leq 2^{-(gcl(x,y)+1)},$$

which is true as $k \geq 1$.

□

Definition 12 (Open balls and open sets)**.** In a metric space (Ω, d), an *open ball* around $x \in \Omega$ with radius $r > 0$ is the set $B(x,r) = \{y \in \Omega : d(x,y) < r\}$. An *open set* is a set A with the property that for every point $x \in A$ there exists an open ball around x that is a subset of A.

For example, in $\mathbb{R}$ the open balls are exactly the open intervals $]a,b[, a < b$. Furthermore, the open balls of (Σ^∞, d_C) are exactly the basic cylinders from above as we show now.

Theorem 3. *The set of open balls in (Σ^∞, d_C) and the basic cylinders coincide.*

Proof. To see this, pick some open ball b. There exists a unique stream s and some radius r such that $b = B(s,r)$. Let $k = \lfloor \log_2(\frac{1}{r}) + 1 \rfloor$. The streams in b are the streams that have a common prefix with s of size at least k. Hence,

$$B(s,r) = \{x \in \Sigma^\infty : s{\downarrow_k} \sqsubset x\} = \mathcal{C}(s{\downarrow_k})$$

and thus every open ball is equal to a basic cylinder. On the other hand, let a basic cylinder $\mathcal{C}(x)$ be given with $x \in \Sigma^k$. Let $y \in \Sigma^\infty$ with $x \sqsubset y$ be an arbitrary extension of x. Then the open ball $B(y, 2^{-(k-1)})$ defines the streams that share a common prefix with y of length at least k. As x is the prefix of y with length k, it follows that $\mathcal{C}(x) = B(y, 2^{-(k-1)})$. Thus every basic cylinder is equal to an open ball. □

The following definition introduces a property of subsets in a metric space, which we will use directly in the next definition of separable spaces, which are a special type of metric spaces.

Definition 13 (Dense subset)**.** A set A is *dense* in a set X if, for all $x \in X$ and $\epsilon > 0$, there exists $a \in A$, such that $d(x,a) < \epsilon$.

Definition 14 (Separable space)**.** A metric space is called *separable* if it contains a countable, dense subset.

Every metric space also defines a σ-field, the so called *Borel σ-field*, which is generated by the open sets of the metric space. It is well-known that for a separable metric space, its associated Borel σ-field is already generated by the open balls. For a metric space X we denote its associated Borel σ-field with $\mathcal{B}(X)$. A function $f : X \to Y$ is called *Borel-measurable* if it is $(\mathcal{B}(X), \mathcal{B}(Y))$-measurable. The analogy of measurable functions for metric spaces are continuous functions:

Definition 15 (Continuous function)**.** A function f between two metric spaces (X,d) and (X',d') is said to be *continuous* in x if for every $\epsilon > 0$ there exists a $\delta > 0$ such that $d'(f(x), f(x')) < \epsilon$ for all x' with $d(x,x') < \delta$. It is said to be continuous if it is continuous in all x.

We are interested in Borel σ-fields as they provide a nexus between continuity and measurability. This relationship is expressed in the following well-known theorem.

Theorem 4. *Every continuous function is Borel-measurable.*

To apply the above theorem to our setting we have to show that the σ-field $\mathcal{S}$ is in fact the Borel-σ-field associated with (Σ^∞, d_C). We already showed that the basic cylinders generating $\mathcal{S}$ coincide with the open balls of our metric space. Hence, the only thing left to do, is to prove that (Σ^∞, d_C) is separable.

Theorem 5. *The metric space (Σ^∞, d_C) is separable.*

Proof. Choose an arbitrary element e of Σ and $D = \{x \cdot y : x \in \Sigma^* \wedge y = \langle e, e, \ldots \rangle\}$. The set D is at most countable. Now pick an element s of Σ^∞ and some $\epsilon > 0$. We choose $k > \log_2(\frac{1}{\epsilon})$ and construct $d = s{\downarrow_k} \cdot \langle e, e, \ldots \rangle$. As d and s have a common prefix of at least k by construction,

$$d_C(s, d) \leq 2^{-k} < 2^{-\log_2(\frac{1}{\epsilon})} = \epsilon \ .$$

As $d \in D$ the set D is dense in Σ^∞. □

Corollary 1. *The σ-field $\mathcal{S}$ generated by basic cylinders of streams, is the Borel σ-field with respect to (Σ^∞, d_C).*

And therefore we can conclude:

Corollary 2. *Every continuous function $\Sigma^\infty \to \overline{\mathbb{R}}$ is also $(\mathcal{S}, \mathcal{B}(\overline{\mathbb{R}}))$-measurable.*

4.1.2 System Modeling Theory

In this section we introduce, Focus (Broy, 2010b; Broy and Stølen, 2001), a mathematical theory for modeling software-intensive systems. Focus provides a precise notion of a system and a formal underpinning of the notions of interface, behavior and composition. The presentation in this section follows mostly Broy (Broy, 2010a; Broy and Stølen, 2001) and Neubeck (Junker and Neubeck, 2012; Neubeck, 2012).

Syntactic Interface

Systems may communicate with their environment by exchanging *messages* on *channels*. The input and output channels of system form its *interface*. We represent a channel by an identifier from the global set of channel identifiers $\mathbb{I}$. A channel can transmit messages of a certain *type*. A type is a subset of the global universe of values $\mathbb{U}$. A channel is assigned a type by the function

$$\mathsf{type} : \mathbb{I} \to \wp(\mathbb{U}) \ .$$

As we deal with discrete systems, we assume that $\mathsf{type}(c)$ is at most countably infinite for each channel.

Definition 16 (Syntactic Interface)**.** The *syntactic interface* of a component is represented by a pair $I \triangleright O$ of (countable) channel sets denoting input and output channels respectively.

Example 1. We consider the example of a data transmission component (TMC) from Broy and Stølen (2001) as a running example. The task of the TMC is to relay the messages it receives on its input channel i via its output channel o. The syntactic interface of the TMC is hence $\{\mathsf{i}\} \triangleright \{\mathsf{o}\}$. The type of the channels is some arbitrary type $T \subset \mathbb{U}$. Figure 4.1 visualizes the syntactic interface of the TMC.

Interface Behavior

The behavior of a system is expressed by relating the incoming messages and the outgoing messages. The assignment of messages to channels is given by a channel valuation.

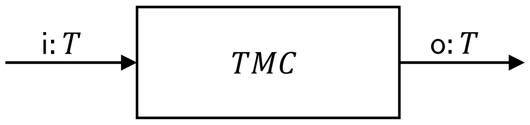


Figure 4.1: Syntactic interface of the TMC

Definition 17 (Channel Valuation)**.** A *channel valuation* for a set of channels C is a mapping

$$v : C \to \mathbb{U} \cup \{\square\} \ ,$$

where $\square$ denotes "no message". If a channel carries a message, the message is required to have the correct type:

$$\forall c \in C : v(c) = \square \vee v(c) \in \mathsf{type}(c) \ .$$

The set of all channel valuations for channels C is denoted by $\overline{C}$. For the empty channel set $\varnothing$ the corresponding set of valuations $\overline{\varnothing}$ contains only the empty valuation, denoted by $\mathbf{0}$. We can restrict a channel valuation to smaller sets of channels. Given a channel set $C' \subseteq C$, and a valuation $v \in \overline{C}$, the *restriction* of v to channels in C' is the valuation $v|_{C'} \in \overline{C'}$, defined by

$$\forall c \in C' : v|_{C'}(c) = v(c) \ .$$

By forming streams of channel valuations we model the flow of incoming and outgoing messages over channels.

Definition 18 (Communication History)**.** A *communication history* for a set of channels C is an infinite stream $\langle v_0, v_1, \ldots \rangle$ of channel valuations $v_i \in \overline{C}$. We denote the set of all communication histories for channel set C with $\overline{C}^{\infty}$ and use $\mathbf{0}$ also for the unique history over the empty channel set.

The restriction operation can be lifted to a communication history h and to a measure over communication histories μ. We will write $h|_C$ and $\mu|_C$ to denote these restrictions. We can lift the operation even further to sets of these structures and will apply the $|$-operation accordingly.

The behavior that is visible at the interface of a system (i.e. the output histories that are produced as reactions to input histories) is called the system's *interface behavior*. Generally, we can distinguish different kinds of interface behavior along the dimensions determinacy and probabilistic nature. In this thesis, we employ the rather general model of interface behavior introduced by Neubeck (2012) that allows to capture non-deterministic as well as probabilistic phenomena. We model such behavior as a *behavior function*.

Definition 19 (Behavior function)**.** A *behavior function* is a function

$$B : \overline{I}^{\infty} \to \wp(\mathbf{Pr}(\overline{O}^{\infty}))$$

from input histories to sets of probability measures over output histories. We will sometimes use the term *component* as a synonym to behavior function.

We demand from a behavior function to be *causal*. Causality enforces that a function adheres to the flow of time. That means a certain output at a specific point in time may only depend on inputs that have already been received up to this point in time. Let $\mu{\downarrow_n}$ be the probability distribution derived from a probability distribution μ by the random variable $(\cdot){\downarrow_n}$. The function B is called *causal*, if

$$\forall h, h', n : h{\downarrow_n} = h'{\downarrow_n} \Rightarrow \{\mu{\downarrow_n} : \mu \in B(h)\} = \{\mu{\downarrow_n} : \mu \in B(h')\} \ .$$

We denote the set of all causal behavior functions with $[I \triangleright O]$. We call a behavior function F *non-probabilistic*, if it only produces measures that assign either probability 1 or 0 to each singleton set:

$$\forall i \in \overline{I}^{\infty}, o \in \overline{O}^{\infty} : F(i)(\{o\}) \in \{1, 0\} \ .$$

We can represent such a behavior by an equivalent function $F^N : \overline{I}^{\infty} \to \wp(\overline{O}^{\infty})$, according to the equation

$$o \in F^N(i) \Leftrightarrow F(i)(\{o\}) = 1 \ .$$

By $[I \overset{n}{\triangleright} O]$ we denote the set of non-probabilistic behavior functions. We furthermore call a behavior function F *deterministic*, if it only produces exactly one probability measure for each input, in symbols:

$$\forall i \in \overline{I}^{\infty} : |F(i)| = 1 \ .$$

We can represent a deterministic behavior function by a function $f : \overline{I}^{\infty} \to \mathbf{Pr}(\overline{O}^{\infty})$ according to the equivalence

$$f(i) = \mu \Leftrightarrow F(i) = \{\mu\} \ .$$

With $[I \overset{d}{\triangleright} O]$ we denote the set of all deterministic behavior functions with syntactic interface $I \triangleright O$. We can characterize a behavior function by its *deterministic realizations*. A deterministic realization of a behavior function F is a deterministic behavior function f that conforms to the specification given by F. For a given behavior function F we obtain the set of all its realizations by

$$[\![F]\!] = \{f \in [I \overset{d}{\triangleright} O] : \forall i \in \overline{I}^{\infty} : f(i) \in F(i)\} \ .$$

If a system has no input channels, there is only one possible input to its behavior function, the unique history consisting of empty valuations, denoted by $\mathbf{0}$. As it is unique, the following definition is often convenient.

Definition 20 (Execution)**.** For a behavior function B without input channels, we define its *execution*

$$\langle\!\langle B \rangle\!\rangle = B(\mathbf{0}) \ .$$

Composition

It is good engineering practice to break a large problem down into smaller problems, solve the smaller problems, and integrate the solutions. Focus supports this paradigm, therefore, systems can be composed of subsystems (or subfunctions), enabling a modular description of a system. The structuring of a system in its constituents is called an *architecture*. Formally, the composition of subsystems is represented in Focus by a binary operator $\otimes$ on behavior functions, called *composition operator*. Before we give a definition of the composition operator, we discuss the concepts *statistical independence* and *information hiding*.

When composing two behavior functions, we have to deal with the issue of statistical dependency. This issue arises when we build the common probability space from the associated probability spaces of the individual behavior functions. If the outputs of the two functions are *statistically independent*, the common probability space can be uniquely constructed. The probability of two outputs occurring together is then the product of the single probabilities. In the case where statistical independence cannot be assumed, a single joint probability measure cannot be constructed. Instead, the result is the set of all possible joint measures that agree with the individual measures.

Information hiding is a general principle supposed to increase the maintainability of system designs by hiding internal information. Applying this to our situation, when composing two subsystems, the channels used for communication only between the subsystems should be hidden to the outside in the composed system. This is known as *channel hiding*.

For this thesis, we will assume statistical independence for the composition. It is possible to formally define the composition without statistical independence. For details on this, see Neubeck (2012). Our definition of the composition does not include channel hiding. Instead, we provide an explicit operator $\dagger$, to realize channel hiding. Given two behavior functions $A \in [I_A \triangleright O_A]$ and $B \in [I_B \triangleright O_B]$. We call A and B *composable*, if the sets of output channels are disjunct: $O_A \cap O_B = \varnothing$. We first define the composition for deterministic behavior functions and use this for the general composition.

Definition 21 (Independent, Deterministic Composition)**.** Given two composable deterministic behavior functions $a \in [I_A \overset{d}{\triangleright} O_A]$ and $b \in [I_B \overset{d}{\triangleright} O_B]$. We define channel sets $O = (O_A \cup O_B)$ and $I = (I_A \cup I_B) \setminus O$. The statistically independent composition $(a \otimes b) \in [I \overset{d}{\triangleright} O]$ is given by

$$\forall z \in \overline{I \cup O}^{\infty}, t \in \mathbb{N} :$$
$$\mathbf{Pr}[(a \otimes b)(z|_I) \sqsupset z|_O{\downarrow}t] = \mathbf{Pr}[a(z|_{I_A}) \sqsupset z|_{O_A}{\downarrow}t] \cdot \mathbf{Pr}[a(z|_{I_B}) \sqsupset z|_{O_B}{\downarrow}t] \; .$$

Note, that we used the convience notation introduced in Section 4.1.1 for the above definition. We can now define the general composition of non-deterministic behavior functions in terms of their deterministic realizations.

Definition 22 (General Composition)**.** Given two composable behavior functions $A \in [I_A \triangleright O_A]$ and $B \in [I_B \triangleright O_B]$. We define channel sets $O = (O_A \cup O_B)$ and $I = (I_A \cup I_B) \setminus O$. The composition $(A \otimes B) \in [I \triangleright O]$ is given by

$$(A \otimes B)(i) = \{(a \otimes b)(i) \, : \, a \in [\![A]\!] \wedge b \in [\![B]\!]\} \; .$$

In order to realize information hiding (via channel hiding), we introduce the projection of a behavior to a subset of its output channels.

Definition 23 (Projection)**.** Given a behavior function $F \in [I \triangleright O]$ and a channel set $O' \subseteq O$, the *projection* of F to the output channels O' is the behavior $F \dagger O'$ defined by

$$\forall i \in \overline{I}^{\infty} : (F \dagger O')(i) = F(i)|O'$$

We can apply the projection to hide internal communication channels. For two behavior functions $A \in [I_A \triangleright O_A]$ and $B \in [I_B \triangleright O_B]$ with internal channels $C = (I_A \cap O_B) \cup (I_B \cap O_A)$ we can therefore specify the composition with channel hiding as $(A \otimes B)\dagger C$. We denote a composition (possibly with projection) graphically by connecting common channels and drawing a dashed border to distinguish hidden channels. See Figure 4.2 for an example of the graphical representation of a composition of two behavior functions A and B. We call such a graphical representation a *component diagram* or *data-flow network*.

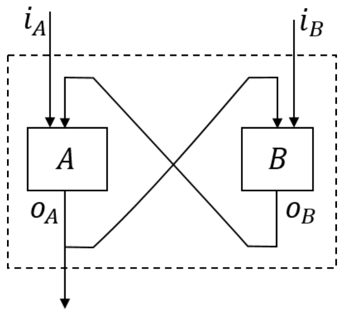


Figure 4.2: Graphical denotation of composition and projection of behavior functions A and B with hidden channel O_B.

4.1.3 Description Techniques

Throughout the thesis we use three different techniques to describe concrete behavior functions: logical interface assertions, state transition diagrams and I/O tables. We will always embed a behavior description into a specification frame to denote the syntactic interface. A specification frame includes a specification name, optionally parameters (such as constants or types) and the input and output channels together with their types. On the upper right side of a specification frame a label may be annotated sometimes to clarify the semantics of the specification. The specification frame below specifies the interface $I \rhd O$ with $I = \mathsf{i}_1, \dots, \mathsf{i}_n$ and $O = \mathsf{o}_1, \dots, \mathsf{o}_n$. The channels are of types T_k^i and T_l^o respectively. The specification is parameterized with constants and types.

name (**const** $c_1,\dots$, **type** $T_1,\dots$) Label

in $\mathsf{i}_1 : T_1^i, \dots, \mathsf{i}_n : T_n^i$

out $\mathsf{o}_1 : T_1^o, \dots, \mathsf{o}_m : T_m^o$

Interface Assertions

An interface assertion is a logical formula with channel names as free variables, ranging over streams of messages taken from the according type. We use interface assertions only to specify non-probabilistic behavior. An interface assertion for an interface $I \rhd O$ with $I = \{\mathsf{i}_1, \dots, \mathsf{i}_n\}$ and $O = \{\mathsf{o}_1, \dots, \mathsf{o}_m\}$ is given by a formula Φ with free variables $i_1, \dots, i_n, o_1, \dots, o_m$. This formula induces a non-probabilistic behavior function $F \in [I \overset{n}{\rhd} O]$ according to the equation

$$\forall i \in \overline{I}^{\infty}, o \in \overline{O}^{\infty} : o \in F(i) \Leftrightarrow \Phi[i_1 \mapsto i|_{\mathsf{i}_1}, \dots, i_n \mapsto i|_{\mathsf{i}_n}, o_1 \mapsto o|_{\mathsf{o}_1}, \dots, o_m \mapsto o|_{\mathsf{o}_m}] \,.$$

Example 2 (Interface assertion for TMC)**.** As an example we specify the behavior of the TMC according to Broy and Stølen (2001). Below, $i \sim o$ means that o is a permutation of i, defined by

$$\forall m \in T : \{m\} \circledS i = \{m\} \circledS o.$$

TMC

in $\mathsf{i} : T$
out $\mathsf{o} : T$

$i \sim o$

State Transition Diagrams

A system's behavior can be understood in terms of its internal states and the transitions between these states. More formally, we can describe a system behavior with a *non-deterministic, probabilistic I/O automaton*. Such an automaton is represented by a tuple $M = (S, I, O, \delta, \Delta, \omega)$, where

- S is the set of states,
- I is the set of input channels,
- O is the set of output channels,
- $\delta \in \wp(\mathbf{dist}(S))$ is a set of initial distribution,
- $\Delta : S \times \overline{I} \to \wp(\mathbf{dist}(S))$ is a non-deterministic transition function, and
- $\omega : S \to \overline{O}$ is an output function.

From an automaton M a behavior function $F_M \in [I \rhd O]$ can be derived (*behavior abstraction*). See Neubeck (2012) for details on how this abstraction is obtained. This motivates using *state-transition diagrams* as description technique for behavior functions. A specification of a non-deterministic, probabilistic automaton includes

- the *syntactic interface*,
- a specification of *local state variables*,
- named *control states* with outputs statements, and
- *transitions* annotated with preconditions, input bindings, postconditions and probability values.

A detailed treatment of state transition diagrams as specification technique can be found in Broy and Stølen (2001) and, for the probabilistic case, in Neubeck (2012). See Figure 4.3 for an example of a state-transition specification. In this example, we model a biased coin tossing. The specification is parameterized by K, the number of consecutive coin tosses. From an initial state, after receiving the start signal S, the coin goes either in state for head (H) or tail (T). Accordingly, either H or T is written to the output channel. However, the coin is biased: the probability distribution for each coin toss may vary non-deterministically by 0.01 around the ideal uniform distribution with probability of $\frac{1}{2}$ for both head and tail. This is represented in the diagram through the probability range $(0.49, 0.51)$ at the transition into the head and tail states. The local variable c serves as toss counter.

Coin $(\textbf{const}\ K : \mathbb{N})$

in $\mathsf{i} : \{\mathsf{S}\}$
out $\mathsf{o} : \{\mathsf{H}, \mathsf{T}\}$
local $c : \{1, \ldots, K\}$ **initial** K

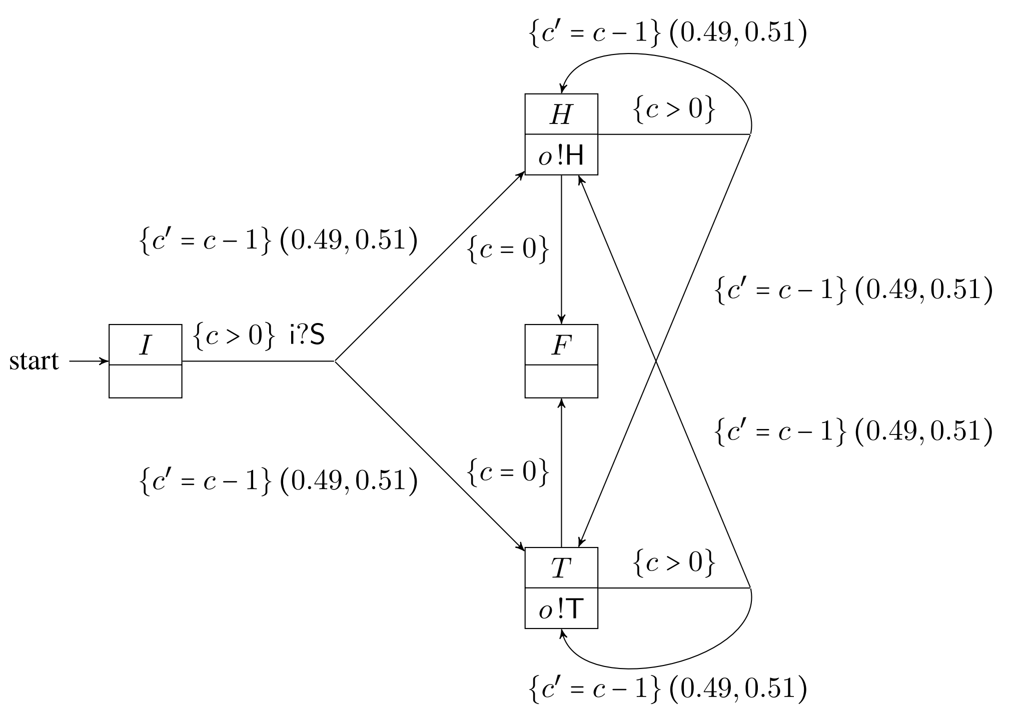

$\{c' = c - 1\}\ (0.49, 0.51)$
H
$o\,!\mathsf{H}$
$\{c > 0\}$
$\{c' = c - 1\}\ (0.49, 0.51)$
$\{c = 0\}$
$\{c' = c - 1\}\ (0.49, 0.51)$
start
I
$\{c > 0\}$ $\mathsf{i}?\mathsf{S}$
F
$\{c' = c - 1\}\ (0.49, 0.51)$
$\{c' = c - 1\}\ (0.49, 0.51)$
$\{c = 0\}$
T
$o\,!\mathsf{T}$
$\{c > 0\}$
$\{c' = c - 1\}\ (0.49, 0.51)$

Figure 4.3: Probabilistic state-machine describing K-times throwing a biased coin.

I/O Tables

We furthermore describe behavior using *input/output tables* (I/O tables). These tables can be employed to describe both, probabilistic and non-probabilistic, behavior. A detailed discussion of the usage of tables for the description of probabilistic behavior can be found in the publication by Broy and Stølen (2001) and Neubeck (2012).

A specification based on I/O tables includes in the frame header the syntactic interface, a declaration of local variables and a declaration of universal variables. The table as such consists of several input columns (one for each input channel and each local variable) and several output columns (one for each output channel and local variable). We use the convention that, for a local variable v, the variable v' refers to the value of v in the next step. In case of probabilistic behavior, an additional column carries the probability of choosing a particular row.

We only describe its semantics informally. For a formal semantics, see Broy and Stølen (2001) and Neubeck (2012). In each step, all rows are matched against the current values of local variables and the input channels. A row matches, if, for some valuation of the universal variables, the values of the local variables and inputs equal the expression in the respective column of the row. From the matching rows, one row is chosen according to the probabilities in the last column. The outputs and local variables for the next step are set according to the expressions in the according columns. If the behavior is not probabilistic and more than one row matches, a row is chosen non-deterministically. Below is the general scheme of an I/O table specification. In the specification, by writing $t[u_1, \ldots, u_k]$, we stress that term t potentially refers to variables $u_1, \ldots, u_k$.

name (parameters)

in $i_1 : T^i_1, \ldots$
out $o_1 : T^o_1, \ldots$
local $l_1 : T^l_1, \ldots$
univ $u_1 : T^u_1, \ldots$

Input				Output				
i_1	$\ldots$	l_1	$\ldots$	o_1	$\ldots$	l'_1	$\ldots$	Prob
				$\vdots$				
$t^i_{n,1}[u_1,\ldots]$	$\ldots$	$t^l_{n,1}[u_1,\ldots]$	$\ldots$	$t^o_{n,1}[u_1,\ldots]$	$\ldots$	${t^l_{n,1}}'[u_1,\ldots]$	$\ldots$	p
				$\vdots$				

4.1.4 Artifact Model

Often, we describe a system with several different models. Each model focuses on specific aspects of the system on a specific level of abstraction. To structure the different models and to describe their roles and relationships we use an *artifact model*. An artifact model is a meta-model, specifying the abstract content (i.e. the models) contained in a certain *artifact* and describes relationships between these models.

As a foundation for this thesis, we employ an artifact model developed by Broy (2011) and Vogelsang (2015). A similar artifact model has been used in the research project SPES (Pohl et al., 2012). See the illustration in Figure 4.4, which is taken from Vogelsang (2015), for an overview over this artifact model. For this thesis, we focus on the following three artifacts:

- The *functional architecture* describes the structure and the behavior of the system from a black-box perspective (i.e. at the system interface). The system is structured into *system functions*. A system function relates to the observable system behavior at a part of the system interface. The structuring is guided primarily by functionality. No specific technical or logical solution is prescribed. The functions of a functional architecture may be structured flatly in a *function list* or hierarchically in a *function hierarchy*. A functional architecture also describes how functions interact by exchanging data. Typically, there are rather few channels between functions and most of them denote *modes* of operation (Vogelsang, 2015). A function hierarchy can be expressed in the formal framework of Section 4.1: A system function f is represented by a behavior function F. This behavior function is defined by the composition of the behaviors of all subfunctions $sub(f)$ of f. This is expressed formally by
$$F = \bigotimes_{f_s \in sub(f)} f_s \ .$$
An extensive treatment of functions and function hierarchies based on Focus can be found in the work by Broy (2010b).

- The *logical component architecture* is also a description of the behavior and the structure of the system. However, the system is not decomposed into system functions, but is structured into logical components in a *component model*. The decomposition is no longer guided purely by functional criteria. Additionally, concerns such as maintainability, safety, and availability are considered. Furthermore, the logical architecture is a first step towards the technical platform on which the system will be deployed. Components on the logical level may form complex data-flow networks. Not every component needs to take part in the externally visible system interface. The component model of the logical architecture can also be represented within our formal framework. A component c is represented by a behavior function C and the behavior of the whole architecture is given by the composition of the behavior functions for all components.

- The *technical architecture* describes the physical devices such as ECUs (Electronic Control Unit) and busses on which the system will be executed. The *platform model* describes these devices and how they interact, similar to the component model in the logical architecture. Additionally, the *deployment model* describes which of the components in the logical architecture are supposed to be executed on which components in the platform model. Like in the logical component architecture, the technical components have a behavior. For this thesis, we assume that the behavior is given by the behavior of logical components deployed on the devices.

The models of the different artifacts are not independent from each other but are related in various ways. For example, the interface behavior of the system as described in the logical component architecture is a refinement of the interface behavior described in the functional architecture.

In Chapter 5, we will build on the artifact model outlined above and extend it by specific models in order to include availability issues.

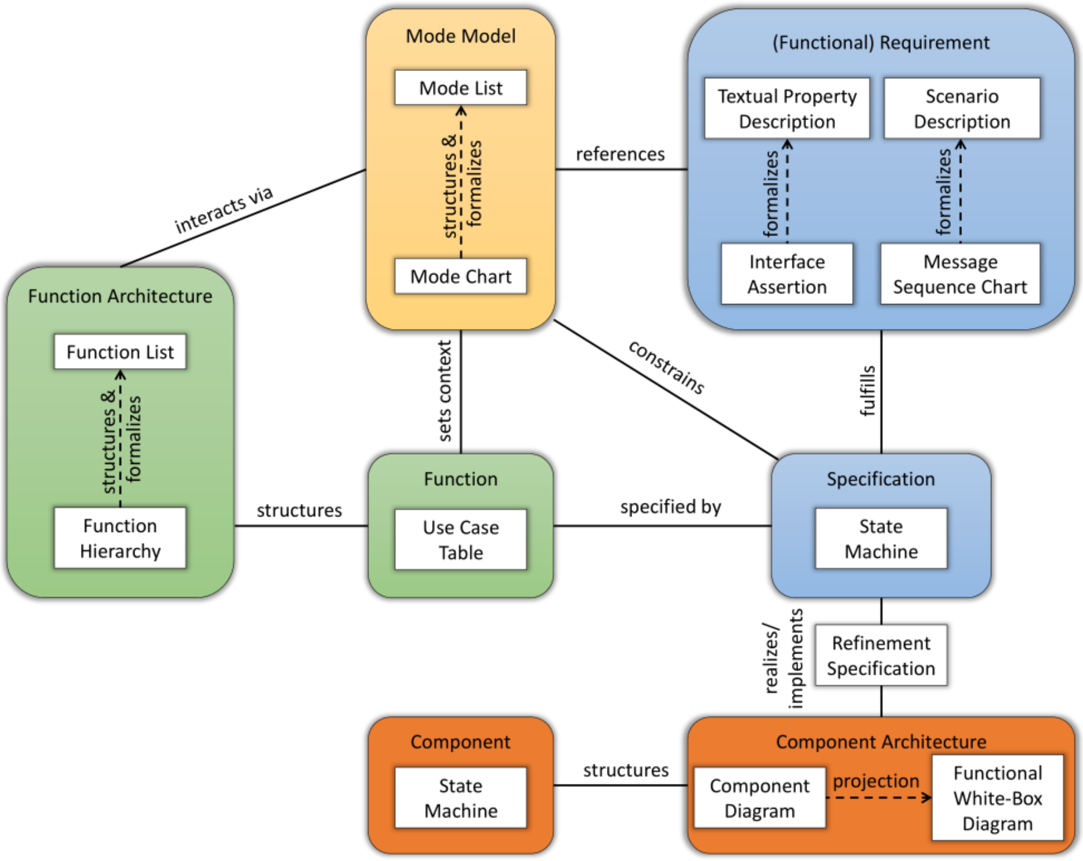


Figure 4.4: Requirements engineering artifact model described in Vogelsang (2015).

4.1.5 Modeling Behavior Deviations

So far, we assumed that we only model the nominal behavior of a system. However, for many purposes it is important to also consider the defective behavior of a system. For this, we want to model deviations of the system's nominal behavior. In the context of FOCUS, approaches for modeling behavior deviations have been proposed by Breitling (2000, 2001) and by Botaschanjan and Hummel (2009). Below, we discuss both and describe a synthesis of them, used for this thesis.

Approach by Breitling (Breitling, 2000, 2001)

Breitling suggests an approach based on FOCUS to model failures based on *modifications* of a nominal behavior (Breitling, 2000, 2001). He considers modifications of specifications with respect to interface behavior, state transition and subsystem architecture. Breitling captures the black-box behavior for the interface $I \triangleright O$ by a relation over input and output histories $S \subseteq I \times O$. The modification of a black-box interface behavior is therefore given by a pair $\mathcal{M} = (E, F)$ consisting of the sets of histories E and F. For a behavior S the modified behavior is then given by

$$S\Delta\mathcal{M} = (S \setminus E) \cup F.$$

If the black-box behavior of the system is specified using an interface assertion Φ, the modification can be expressed using the pair of interface assertions $\mathcal{M} = (\Phi_E, \Phi_F)$. Similar as above, the modified specification is given by

$$\Phi\Delta\mathcal{M} = (\Phi \wedge \Phi_E) \vee \Phi_F.$$

Breitling further defines modifications on state transition systems by adding and removing transitions. Finally, Breitling introduces modification components. These are components that are composed with a given system or component in order to achieve the modification. The modification components act as filters modifying either the input to the system or the output from the system. Breitling distinguishes four patterns how the resulting data-flow network looks like, depending on the role of the modification components as pre-filter or post-filter and depending on the information about the original inputs that the filter components receive (see Figure 4.5). As Breitling notes, the configuration with just a post-filter that, however, receives the original input, is enough to model any modification.

Approach by Botaschanjan and Hummel (Botaschanjan and Hummel, 2009)

Botaschanjan and Hummel take up the idea of modification components from Breitling. Compared to Breitling they add the notion of *error modes*, which represent different types of deviation. While in Breitling's work, a modification is always applied to the whole behavior, Botaschanjan and Hummel provide means to activate or deactivate error modes non-deterministically, triggered by inputs, triggered by other external error coordination signals, or triggered by time. Figure 4.6 shows the extended pattern of modification components by Botaschanjan et al. Error modes are represented by pairs of input and output filter components $(if_1, of_1), \ldots, (if_n, of_n)$. An error mode function decides at each point in time, which (if any) error mode should be activated. The selector components are stereotypical components that select the output of the activated error mode filters. Note that every faulty behavior that can be described with the means of Botaschanjan and Hummel can also be described via Breitlings modification components.

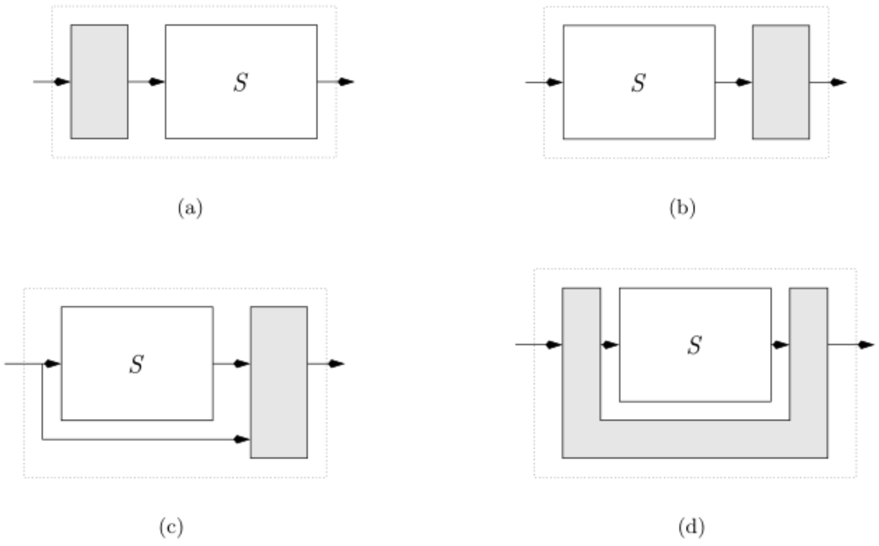


Figure 4.5: Possible configuration of modification components according to Breitling (2000) (Illustration taken from the original publication).

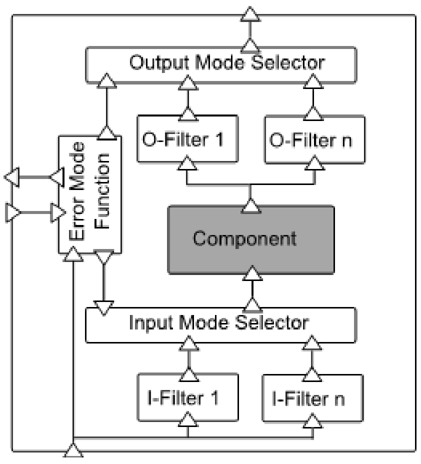


Figure 4.6: Schematic overview over the approach by Botaschanjan and Hummel (2009) (Illustration taken from the original publication).

Synthesis

For our purpose, we adopt an approach similar to the approaches by Breitling and Botaschanjan and Hummel. We call a model that describes deviating behavior with respect to a nominal behavior, a *deviation model*. We use input and output *filters* to model the modification of a given behavior. These filters are activated and deactivated by a dedicated *activation function*.

For the following, let the behavior function F' be the behavior function obtained from function F by renaming all its channels to their primed version. Similar, let the channel sets I' and O' be the channel sets that contain the primed versions of all channels in I and O respectively.

Definition 24. A *deviation model* $\mathcal{D}$ for a behavior function F with syntactic interface $I \triangleright O$ is represented by a tuple (act, if, of), where

- $act \in [I \cup C_I \triangleright C_O \cup \{\mathsf{a}\}]$ is the activation function that defines the activation and deactivation behavior. Its interface subsumes I (the original input channels to F), the channel sets C_I and C_O (input and output channels for the coordination with other deviation models) and a dedicated channel a to communicate the current activation status to the filters.
- $if \in [I \cup \{\mathsf{a}\} \triangleright I']$ is an input filter.
- $of \in [O' \cup \{\mathsf{a}\} \triangleright O]$ is an output filter.

We denote the set of deviation models for interface $I \triangleright O$ with $\mathbb{D}[I \triangleright O]$. With $F \Delta \mathcal{D}$ we denote the *application* of the deviation model $\mathcal{D}$ to the behavior function F defined by

$$F \Delta \mathcal{D} = (F' \otimes act \otimes if \otimes of) \dagger (O \cup C_O)$$

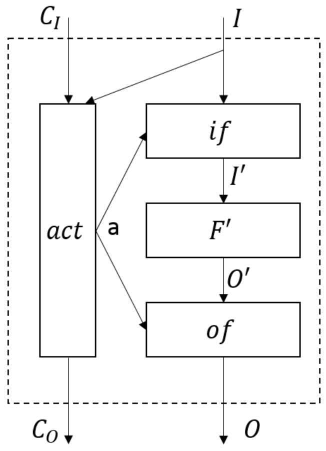
The resulting syntactic interface is $(I \cup C_I) \triangleright (O \cup C_O)$. Several deviation models can be applied in a nested fashion, such as $(F \Delta \mathcal{D}) \Delta \mathcal{D}'$. The data-flow diagram in Figure 4.7a illustrates the semantics of a deviation model.

Description Technique

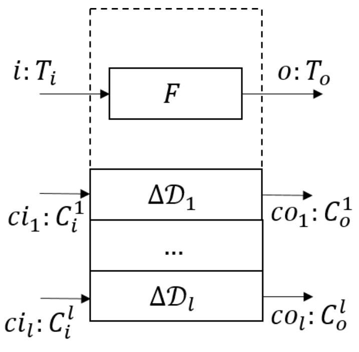
In order to specify a deviation model, we nest the constituents of the deviation model in a common specification frame. We label the activation function with “act”, the input filter with “if” and the output filter with “of”. We label the specification frame with “deviation”. A schematic specification is shown in Figure 4.8. To express the application of deviation models $\mathcal{D}_1, \ldots, \mathcal{D}_l$ to a component C visually, we use the notation illustrated in Figure 4.7b. Here, ci_k and co_k refer to the coordination channels introduced by the deviation models. Furthermore, i and o are the original channels defined by C.

Example

As an example for a deviation model, we formulate a faulty TMC. To achieve the faulty behavior we devise a deviation model that uses an output filter to drop messages. The input filter leaves the input messages untouched. The activation component is described using a probabilistic automaton. The deviation model for the faulty TMC is depicted in Figure 4.9.



(a) Illustration of the semantics of applying a deviation model to a behavior function F.



(b) Visual denotation of the application of deviation models D_k to a behavior function F.

Figure 4.7: Semantics and visual denotation of the application of a deviation model.

dm deviation

act

in $ci_1 : T_{ci}^1, \ldots, ci_n : T_{ci}^n$
out $co_1 : T_{co}^1, \ldots, co_k : T_{co}^k$
a : A

body

if

in $i_1 : T_i^1, \ldots, i_m : T_i^m$
a : A
out $i'_1 : T_i^1, \ldots, i'_m : T_i^m$

body

of

in $o'_1 : T_o^1, \ldots, o'_l : T_o^l$
a : A
out $o_1 : T_o^1, \ldots, o_l : T_o^l$

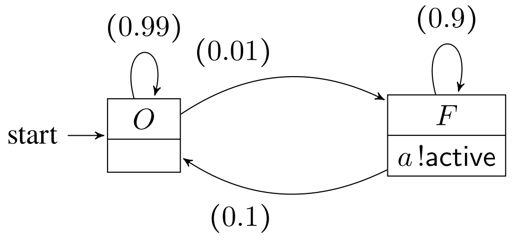
body

Figure 4.8: Schematic structure of a deviation model specification.

DM_TMC — deviation

act

out a : {active}



if

in $i : T$

a : {active}

out $i' : T$

$i = i'$

of

in $o' : T$

a : {active}

out $o : T$

$\forall t \in \mathbb{N}_0 : \ (\mathsf{a}.t = \mathsf{active} \Rightarrow o.t = \square) \wedge (\mathsf{a}.t \neq \mathsf{active} \Rightarrow o.t = o'.t)$

Figure 4.9: Deviation model for the perfect TMC in order to model the faulty TMC.

4.2 Availability: Terms and Definitions

In this section we discuss *availability*, the central topic of this thesis. Availability is being addressed by researchers and practitioners from a broad range of fields and several definitions are in use. However, most of them agree that availability intuitively describes the ability of a system to operate failure-free most of the time. Therefore, availability is closely coupled to the concept of *failure*. A failure occurs if the visible behavior deviates from the specification. There are a number of other terms used in the research community and in practice, which describe similar concepts as availability. Examples are *reliability* or *maintainability*. As this thesis deals exclusively with availability, it is important to distinguish these terms from each other. Availability is not only a property that we analyze after a system is built, but instead we usually demand a certain degree of availability from the system before it is built. This demand is formulated in *availability requirements* during the development of the system.

In this chapter, we start in Section 4.2.1 by explaining the concept of failure and outline how failures emerge from faults and errors. Afterwards, we consider a number of availability definitions from the literature in Section 4.2.2. We limit ourselves to definitions that fit to software-intensive systems. In Section 4.2.3, we distinguish availability from related terms, such as reliability. Finally, in Section 4.2.4, we focus on availability requirements, their role in the system development and which type of content they encompass.

4.2.1 Fault, Error and Failure

Availability is the ability of a system to operate without *failure* most of the time. Failure is therefore an important concept in order to understand availability. Failure, together with the related terms *fault* and *error*, form a chain of causes and effects. A fault refers to an unwanted condition of a piece of hardware or the software source code (or any other software representation). An example for a fault is a wrongly written part of a software (i.e. a software bug) or a piece of hardware that is broken due to wear-out. If the part of the system that is affected by the fault is activated during the course of executing a software intensive system, this may cause an error. An error refers to a state of the system that deviates from a correct state. An example for an error is a wrongly calculated value for an internal variable. Also the runtime program code may be considered part of the state of the system. Finally, a failure is a deviation of the externally visible system behavior from the correct behavior. An error may cause a failure, if the erroneous state (e.g. the wrongly calculated variable) leads to a deviation of the externally visible behavior (e.g. a wrong output value). (Avižienis et al., 2004)

Figure 4.10 summarizes the relationship between the three terms.

Avižienis et al. (2004) further characterize failures by *failure modes*. A failure mode is a specific way a failure manifests itself. For example, consider the transmission example from Section 4.1.2. One failure mode of such a transmission system is the delayed transmission of a message. A second, different failure mode is the modification of the message value. Failures may, furthermore, be classified according to their *severity*. For the delay failure mode in the transmission example, different severity levels could be defined considering the duration of the delay.

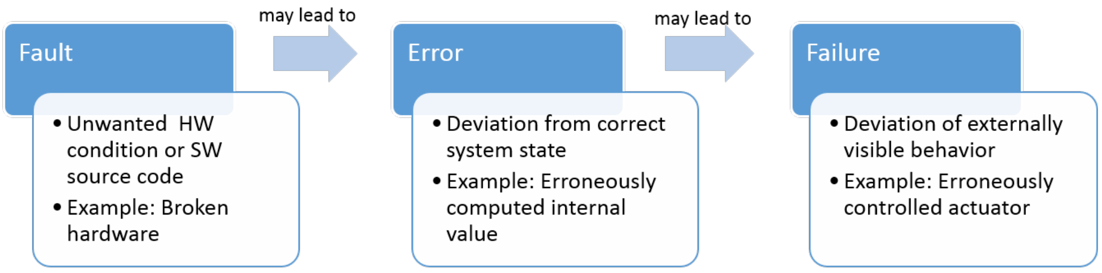

Fault
• Unwanted HW condition or SW source code
• Example: Broken hardware
may lead to
Error
• Deviation from correct system state
• Example: Erroneously computed internal value
may lead to
Failure
• Deviation of externally visible behavior
• Example: Erroneously controlled actuator

Figure 4.10: Characterization and relationship between fault, error and failure.

4.2.2 Availability Definitions from Literature

The term availability has been defined from several different viewpoints in the past. In this section we discuss four types of availability definitions and relate them to the concept of failure introduced in the previous section. First we present the much cited definitions of Laprie and Avižienis from the domain of dependable computing. Second, we consider a definition from ITU-T E.800 recommendation, which is also pervasive in the availability literature. After that we review definitions of *service availability*, a term only recently established. Finally, we discuss availability definitions from the security research community.

The classic definitions of availability in the field of computing are due to Jean-Claude Laprie (Laprie, 1995) and Algirdas Avižienis (Avižienis et al., 2004). Laprie builds his definition on the notion of a *service* and its *specification*. With the term service, Laprie refers to an abstraction of a system's behavior as it is perceived by its users. A *service specification* is the description of the service agreed upon by the supplier of the service and its users. A user interacting with or observing the system perceives the system as "an alternation between two states of the delivered service with respect to the specified service, service *accomplishment*, where the service is delivered as specified, [and] service *interruption* where the delivered service is different from the specified service" (Laprie, 1995). Note, that with service interruption, Laprie refers to the same phenomenon that we termed failure in the preceding Section 4.2.1: the deviation of the externally visible system behavior from the specified behavior. Based on these notions, Laprie defines availability as

> *"a measure of the service accomplishment* with respect to the alternation *of accomplishment and interruption". (Laprie, 1995)*

Laprie leaves open what kinds of measures he refers to in the above definition. However, he stresses that in many cases a probability measure (which is a measure in the mathematical sense) will be employed. Although not explicitly mentioned by Laprie, his definition relates availability to the requirements of the system, as the service specification is derived from the requirements. The definition by Laprie may be considered a definition template, which can be specialized with a concrete measure, a service specification, and a method to distinguish service accomplishment from service interruption.

Avižienis et al. (2004) use a different terminology than Laprie: A *function* in their terminology "is what a system is intended to do". It is described by a *functional specification*. The system behavior is "what the system does to implement its function" and a *service* is the behavior as perceived by the system users. Finally, a system delivers a *correct service* when the service implements its function. Note, that correct service can be identified with the absence of failures in the sense described in Section 4.2.1. With this terminology, Avižienis et al. define availability as

> *"readiness for correct service". (Avižienis et al., 2004)*

Unfortunately the term readiness is not further explained in the publication. This definition also obscures if availability refers to a measure, a property or rather to the state of the system.

A further definition of availability originates from the telecommunication related ITU-T E.800 recommendation (ITU, 2008). It defines availability as the

> *"ability of an item to be in a state to perform a required function at a given instant of time or at any instant of time within a given time interval, assuming that the external resources, if required, are provided". (ITU, 2008)*

As the previous two definitions, it relates availability to the consistency of the behavior with the functional specification by demanding the item to be able to "perform a required function". Hence, also this definition defines availability with the absence of failures. It adds as additional details the reference to a certain time-frame and the precondition of provided external resources.

Although the classic definitions by Laprie and Avižienis et al. already refer to the service delivered by a system *from a user perspective*, the notion of *service availability* has only recently been established. When authors refer to service availability, they usually intend to stress the fact that they take a user's perspective on the availability of a system (cf. Anderson et al., 2001; Rossebeø et al., 2006; Tokuno and Yamada, 2008; Wang and Trivedi, 2005). The according availability definitions are often more specific than the rather generic definitions above. They might be considered as "filling in" the definition template given by Laprie. An example for a definition of service availability in the telecommunication context is given by Wang:

> *"During a user interaction (session) with the system, the user issues multiple requests at different time points for different system resources. [. . .] The service availability is the probability that all requests are successfully satisfied during the user session." (Wang and Trivedi, 2005)*

Finally, the computer security research community takes a different viewpoint on availability. Here, availability is one main security attribute additional to integrity and confidentiality. According to Newman (2009),

> *"Availability means computer and network resources are accessible to only authorized parties." (Newman, 2009)*

From this viewpoint we can distinguish two aspects of availability: accessibility and exclusivity (Rossebeø et al., 2006). Accessibility refers to the ability of the system to grant access to authorized parties while exclusivity guarantees that the system denies access to non-authorized parties.

For the work presented in this thesis we adopt the definition of Laprie: availability as measure of service accomplishment with respect to the alternation of accomplishment and interruption (Laprie, 1995). In our view, its understanding of availability as a measure captures the understanding of availability in practice. It furthermore relates clearly to the functional requirements of the system.

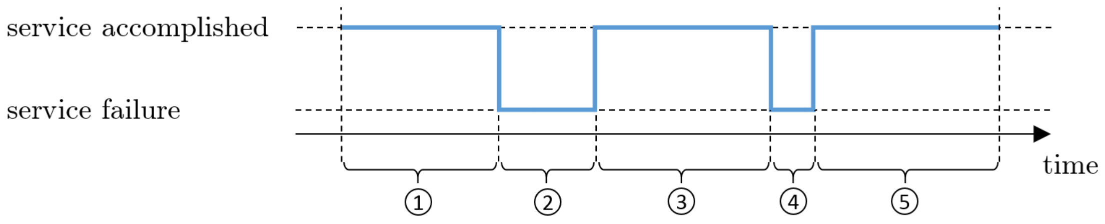


Figure 4.11: Periods of service accomplishment and service failure. Reliability is a measure for period ①. Maintainability relates at least partly to the periods ② and ④. Availability is a measure for the relation between periods ①, ③, ⑤ and periods ②, ④.

4.2.3 Related Terms

In the literature, there are several terms with a similar meaning as availability. All of these terms describe the extend to which a system may exhibit certain types of failures. The research area dependable computing considers the terms *reliability*, *maintainability* and *(functional) safety*. Additionally, as mentioned in the previous section on availability definitions, computer *security* is often related to availability. Finally, especially for network or general communication systems, the term *quality-of-service* is used. Below, we briefly describe these terms and distinguish them from availability.

Reliability

Reliability is the term most closely related to availability. Laprie defines reliability as "a measure of the continuous service accomplishment (or, equivalently, of the time to failure) from a reference initial instant" (Laprie, 1995). Hence, reliability is a measure of the time to the first failure from a given time point, ignoring the behavior of the system after this first failure. Availability, in contrast, is a measure for all the periods where the service is accomplished compared to the periods where the service is interrupted, even after the first failure. Availability, therefore, also considers repair or self-healing, which is ignored by reliability. Figure 4.11 illustrates the difference between reliability and availability. Reliability considers only period ① while availability considers also the later periods.

Maintainability

Maintenance activities of a software system are often broken down into the following four categories (e.g. Bourque and Fairley, 2014):

- *Corrective maintenance:* Repair of a software product, possibly after delivery.
- *Adaptive maintenance:* Modification of a software product to keep it usable.
- *Perfective maintenance:* Modification of a software product to provide enhancements for users or improvements of other attributes of the software.
- *Preventive maintenance:* Modification of a software product to detect and correct latent faults before they become operational.

With these maintenance activities in mind, maintainability means a system's "ability to undergo modifications and repairs." (Avižienis et al., 2004) Especially the ability to perform repairs relates to availability, as fast repairs shorten the periods of service failure (periods ② and ④ in Figure 4.11). However, depending on the definition of service failure, also the ability to perform the other types of maintenance activities may influence availability.

(Functional) Safety

The ISO 26262 norm on the functional safety of road vehicles defines safety as "absence of unreasonable risk" (ISO, 2011) and functional safety as "absence of unreasonable risk due to hazards caused by malfunctioning behavior of E/E systems" (ISO, 2011). According to Avižienis, safety refers to the "absence of catastrophic consequences on the user(s) and the environment" (Avižienis et al., 2004).

Especially functional safety is related to availability: Similar to availability, functional safety refers to malfunctioning behavior (i.e. behavior that is not in line with the functional specification). However, safety focuses on the risk of harming users or the environment that is associated with the malfunctioning. Avoiding situations in which a malfunctioning can cause harm is the goal of functional safety. This focus on avoiding any harm is not connected with availability.

Security

In the computer security research community, security is seen as a compound concept consisting of confidentiality, integrity, and availability (Avižienis et al., 2004; Bishop, 2012; Pfleeger and Pfleeger, 2006). In this context, confidentiality means the absence of unauthorized disclosure of information and integrity means the absence of unauthorized system alterations (Avižienis et al., 2004). A high availability is therefore considered one aspect of security. On the other hand, from an attacker's perspective, reaching the absence of availability is the goal of a specific type of attack on computer systems, causing a so-called *denial of service*.

Quality of Service

Quality of Service (QoS) is a term that is mostly used in conjunction with computer networks or, more generally, communication services. It is often used as an umbrella term for a large set of characteristics of communication services. A central document regarding QoS is the ITU-T E.800 recommendation (Balzer, 2015; ITU, 2008). It defines QoS rather broadly as the "totality of characteristics of a telecommunications service that bear on its ability to satisfy stated and implied needs of the user of the service" (ITU, 2008). However, the recommendation names the following six central quality characteristics for QoS: speed, accuracy, dependability, availability, reliability, simplicity. Hence, similar as for security, availability can be seen as a part of QoS, besides other characteristics.

4.2.4 Availability Requirements

In the following section we consider availability requirements. We first clarify the general *role* of availability requirements and their *sources*. Afterwards, we discuss their *scope* in relation to the artifacts outlined in Section 4.1.4. Next, we outline the contents of availability requirements according to industry standards. Often, availability requirements are stated quantitatively using *availability metrics*. Therefore, in the last part of this section we explain what availability metrics are and give examples.

Role of Availability Requirements

If a system has to achieve a certain availability, this needs to be reflected in the requirements of the system. *Availability requirements* describe the characteristics that a system should possess with regard to availability. As other types of requirements, availability requirements need to be elicited and documented as part of requirements engineering. Typically, according to our interview study and industrial standards, such as IEC 62347 (IEC, 2006), sources for the elicitation of availability requirements are:

- Analysis of the market situation and competitor products
- Customer requirements
- Economic considerations regarding the operation of a system
- Experiences with operation data of legacy systems
- Demands of regulation bodies

As for other types of requirements, the producer of a system needs to provide arguments that the availability requirements are indeed fulfilled in the final product. This argument is usually documented as part of a RAM case. Availability requirements are considered in several industry standards relevant for software intensive systems. Among them are IEC 60300 on reliability management (especially part 3-4, IEC, 1996), IEC 62471 on the demonstration of dependability requirements (IEC, 2012) and IEC 62347 (IEC, 2006) on system dependability specifications.

Scope of Availability Requirements

The scope of availability requirements are usually the *functions* of a system. This means, availability requirements relate to the functions of a system and can be described as soon as the functions have been determined. IEC 62347 outlines a generic process for the system specification and integrates activities regarding the specification of availability requirements into this process. An extract of the process is depicted in Figure 4.12. There are two main activities relevant for availability requirements in this process. First, during the step *Analyze Requirements*, the important dependability characteristics are determined. In the language of IEC 62347, availability is one such dependability characteristic. In other words, during this step, rough availability requirements are determined. The second activity, relevant for availability requirements, is performed during the step *Design and Evaluate Functions*. In this step detailed availability requirements are developed for each system function based on its specification.

Content of Availability Requirements

In Section 4.2.2 we saw that availability is closely related to the system's specification and the definition of what is a failure. Hence, the reference to a system specification and a definition of failure are part of a specification of availability requirements. However, beyond that, an availability requirements specification contains information about the system's environment and how it is maintained. The main reason for this is that the performance of systems can vary, depending on the context in which it is installed and used. Therefore, availability requirements

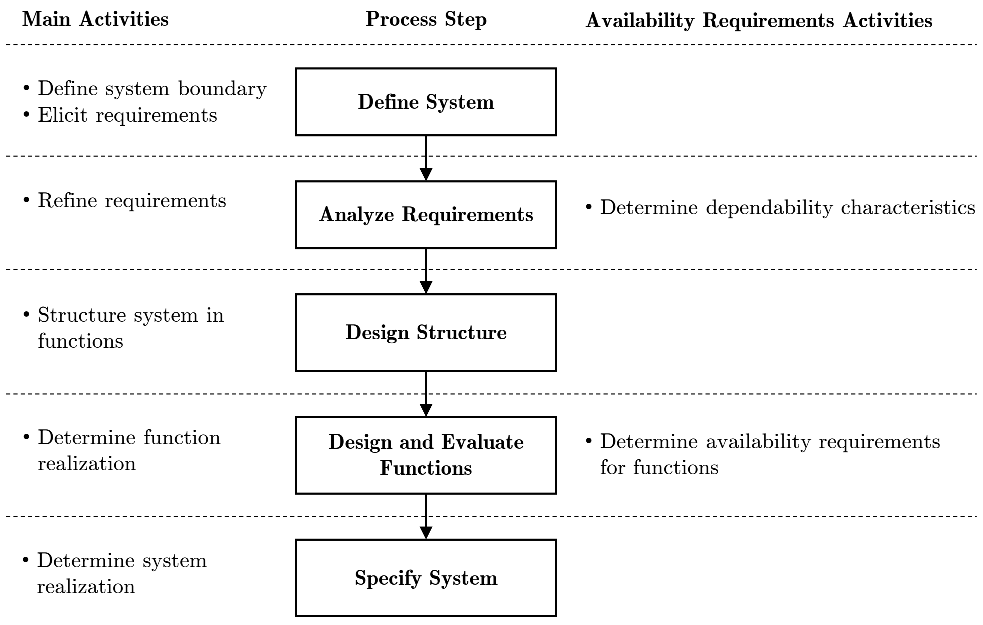

Main Activities
Process Step
Availability Requirements Activities
• Define system boundary
• Elicit requirements
Define System
• Refine requirements
Analyze Requirements
• Determine dependability characteristics
• Structure system in functions
Design Structure
• Determine function realization
Design and Evaluate Functions
• Determine availability requirements for functions
• Determine system realization
Specify System

Figure 4.12: Extract of the process of system specification according to IEC 62347 with activities relevant for availability requirements.

contain a so-called *operational profile* (or *usage profile*) that describe the usage of the system. The IEC 60300-3-4 standard (IEC, 1996), which provides a guideline for the specification of dependability requirements, lists further items that should be contained in a specification:

- system descriptions including system functions,
- environmental conditions and operating conditions,
- failure definition, i.e. a description of the situation when a system (function) is considered failed to which degree,
- information on how the system is installed and used and
- maintenance rules.

The demanded availability of a system can be described *qualitatively* or *quantitatively*, using availability metrics. When availability requirements are expressed quantitatively, IEC 60300-3-3 stresses that the specification needs to state precisely, how the used metrics should be interpreted. That means, for example, which time periods should be counted as service interruption. As availability metrics play an important role for the specification of availability requirements, we discuss them in more detail in the next section.

Availability Metrics

The availability of a system is often described quantitatively. An example for such a quantitative statement on availability is: "The system should have a mean downtime per week of at most 30 minutes". The mean downtime, in this example, is an instance of an *availability metric*. Note, that the term metric has a precise meanings in mathematics (see Section 4.1.1). However, in this thesis, we use the term "availability metric" informally, to relate to any type of quantity that is used to describe availability. Below, we illustrate availability metrics by giving some informal examples. Afterwards, we show, how availability metrics can be understood formally using probability theory. Later on, in Section 5.2.4, we give a more comprehensive formalization of availability metrics in terms of our formal system model.

Examples of Availability Metrics. Kan (2002) and Eusgeld et al. (2008) give examples for different availability metrics. Among them are *uptime*, *downtime*, *frequency of failures* and *duration of failures* together with the related metrics *mean-time-between-failures* (MTTF) and *mean-time-to-repair* (MTTR). Further examples are *number-of-nines*, *point availability* and *steady-state availability*. A domain-specific example of an availability metric for the telecommunication domain is the *dropped call rate*.

Uptime is the time that a system (or system function) is operational during a reference time frame. It can be given as absolute time (e.g. 167 hours per week) or as a fraction of the reference time frame (e.g. 99.5%). *Downtime* is the complement of uptime. It is the time that a system is not operational during a reference time frame. As uptime, it can be given in absolute time (e.g. 50 minutes per week) or as a fraction of the reference time frame (e.g. 0.05%). The *frequency and duration of failures* denote how often a failure occurs during a certain time frame (e.g. two failure periods per week) and how long the these failure periods last (e.g. 30 minutes). Related metrics are the *mean-time-between-failures* (MTBF) which is the expected time between two consecutive failure periods and the *mean-time-to-repair* (MTTR), the expected duration of a failure period. Derived from the percentage representation of uptime, is the metric *number-of-nines*. It refers to the number of nines until the first other digit in these percentage numbers. The fraction of uptime of 99.999% would thus be expressed as “5-nines”. Different terms that are occasionally used for availability levels are “high-availability" (5 nines), “very-high-availability” (6 nines) and “ultra-high-availability” (7 nines) (Gray and Siewiorek, 1991). The *point availability* is a probabilistic metric. It is the probability that the system is operating failure-free at a certain point in time (e.g. 0.999 probability for failure free operation after one week since system installation). *Steady-state availability* is also a probabilistic metric. It is the limiting value of point availability if the time point approaches infinity. As a final example, domain-specific performance metrics are sometimes used to describe a system’s availability. A prominent example comes from the telecommunication domain. The *dropped call rate*, or similar *the defects per minute* (DPM) is an availability metric that counts the number of dropped telephones calls per million calls (Trivedi et al., 2010).

Formal Availability Metrics. A mathematically rigorous definition of availability metrics originates from the field of reliability theory. Reliability theory is a field that has been developing in since the first half of the 20th century and which uses statistical methods for the investigation of reliability and related concepts such as availability (Rausand and Høyland, 2004). Availability metrics here are typically expressed in terms of probabilistic descriptions of failures. Reliability theory distinguishes between discrete time models and continuous time models. As we focus on discrete time systems we will employ the concepts from reliability theory in their discrete versions. In the preceding section we already mentioned *point-availability*. It is given by the probability that an item (such as the system or a system service) is functioning at a certain instant of time (Grottke et al., 2008). In symbols, for a time point $t \in \mathbb{N}$

$$A_p(t) = \mathbf{Pr}[\text{item is functioning at time } t].$$

We can state the proposition “item is functioning at time t” more formally using a random variable $Y(t)$ taking value 1 if the item is operating correctly at time t, and 0 otherwise. Hence, we obtain

$$A_p(t) = \mathbf{Pr}[Y(t) = 1] \; .$$

From this basic definition, more availability metrics can be derived, such as the *uptime* in the discrete time interval $[0;t]$

$$U(t) = \frac{1}{t+1} \sum_{0}^{t} A_p(t) \; ,$$

or the *steady-state availability*, denoting the point availability in the long run

$$A = \lim_{t \to \infty} A_p(t) \; .$$

The steady-state availability can be thought of as the availability after the system has stabilized. After setting the system in operation, there might be short time effects, such as production flaws or unclear maintenance policies, which result in a temporarily higher point-availability. After some time, these effects disappear and the point availability settles at a certain value. This value is the steady-state availability. Consider Figure 4.13 for an illustration of the point-availability approaching the steady-state availability from below.

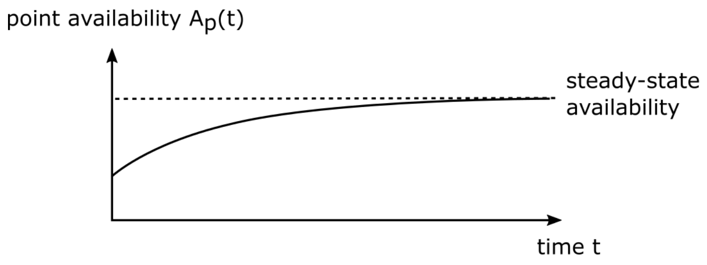


Figure 4.13: Illustration of point-availability approaching steady-state availability.

Chapter 5

Availability Artifact Model

Our approach in this thesis is to capture the information, necessary for the specification and analysis of availability, using specialized models. To guarantee a well-defined semantics of the models, we base them on the common modeling theory Focus (see Section 4.1). To describe and organize the models, and to define the relationships among them, we integrate them into an artifact model. More specifically, we extend the artifact model outlined in Section 4.1.4. This chapter is structured in the following way: First, we give a short overview over the extended availability artifact model in Section 5.1. In the subsequent sections 5.2 to 5.5, we detail our proposed artifacts and the involved models. Finally, in Section 5.6, we describe availability analyses based on the introduced models.

5.1 Overview

Figure 5.1 gives an overview over the availability artifact model. The blue artifacts are parts of the original artifact model, whereas the green artifacts are our extensions and will be in the focus of this chapter. For details on the blue artifacts see Chapter 4. White boxes denote models used as part of an artifact. The arrows denote relationships between artifacts and models. As Figure 5.1 shows, we add four artifacts to the artifact model: The *availability requirements specification* contains the requirements on the system regarding availability. It makes use of the second artifact, the *availability specification*, which contains the necessary definitions of failure and of availability metrics. The *extended logical architecture* is an extension of the original logical architecture to include the behavior in case of faults. Finally, the *environment specification* describes the behavior and structure of external systems and users. Below, we give a short description of all artifacts and models. We will provide the details in the following sections.

Availability Requirements Specification The availability requirements specification captures the demands of the different stakeholders of the system that relate to availability. As outlined in Section 4.2.4, there are various sources for availability requirements, such as the customer, regulation bodies or economic considerations. We capture availability requirements informally with *textual availability descriptions*. These are statements about availability formulated in natural language. They may refer to the functional requirements and to elements of the functional architecture (e.g. availability requirements

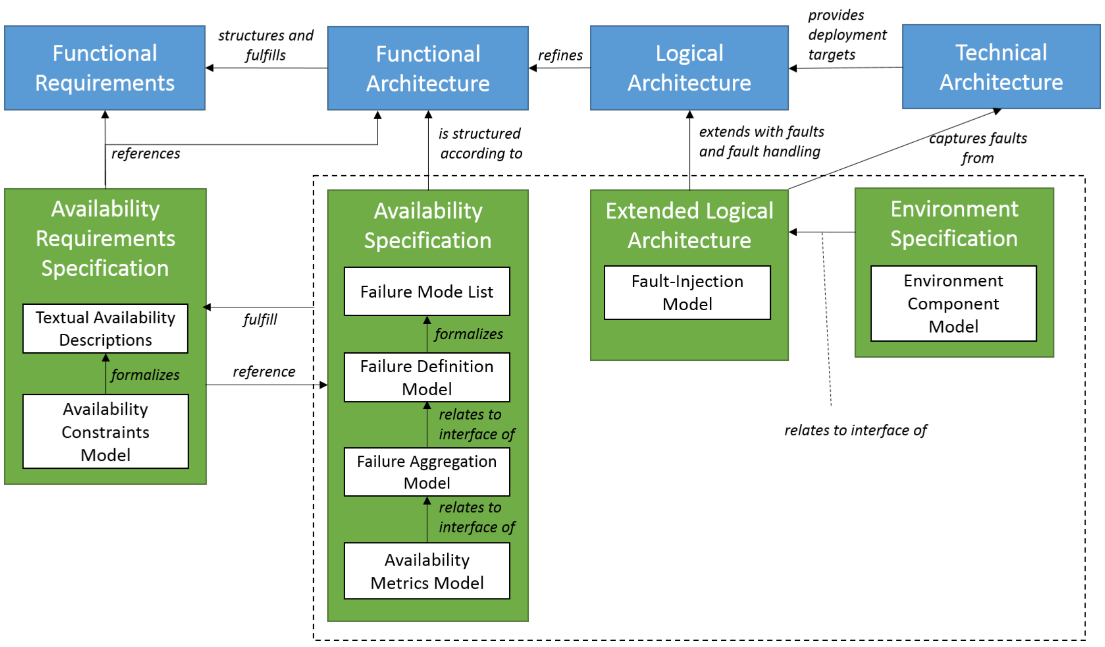

Functional Requirements
structures and fulfills
Functional Architecture
refines
Logical Architecture
provides deployment targets
Technical Architecture
references
is structured according to
extends with faults and fault handling
captures faults from
Availability Requirements Specification
Textual Availability Descriptions
formalizes
Availability Constraints Model
fulfill
reference
Availability Specification
Failure Mode List
formalizes
Failure Definition Model
relates to interface of
Failure Aggregation Model
relates to interface of
Availability Metrics Model
Extended Logical Architecture
Fault-Injection Model
Environment Specification
Environment Component Model
relates to interface of

Figure 5.1: Overview over the Availability Artifact Model. Green boxes relate to artifacts that we focus on in this chapter. Blue boxes related to artefacts introduced before (see Chapter 4). White boxes denote the model types used in these artifacts. Arrows describe the relationship between models and artifacts.

can be formulated for each system function). We formalize these descriptions in an *availability constraints model*. This model represents constraints on availability (such as ">99.999% uptime") as mathematical structures that relate to well-defined formal availability metrics as specified in the availability specification artifact (see relationship *reference* in Figure 5.1). We discuss the availability requirements specification in detail in Section 5.5.

Availability Specification The availability specification provides definitions of what availability should mean for the given system. Especially, it contains a definition of what failure means (i.e. which deviations from the specification are failures) and rules how to calculate availability metrics. It thus provides the basis, on which formal availability requirements are formulated. We use four models to capture and to stepwise formalize the necessary information. The *failure mode list* is a first step towards the definition of what failure should mean. It outlines and describes informally the failure modes, considered relevant to describe the availability of the system at hand. It is structured according to the system functions. The *failure definition model* formalizes and extends the failure mode list. Most importantly, it relates deviations from the system's specified behavior to the failure modes. The *failure aggregation model* is an optional model that aggregates and simplifies failure modes to facilitate the definition of availability metrics. Finally, the *availability metrics model* defines availability metrics by fixing rules how to calculate the metrics. We present details on the availability specification in Section 5.2.

Extended Logical Architecture The *extended logical architecture* extends the original logical architecture. The purpose of this extension is to include the behavior of the system in case of faults. Only when faults and their effect on the system behavior are modeled, we can perform meaningful availability analyses. The only model in this artifact is the *fault-injection model*. This model includes, potentially for every component in the logical architecture, a description of how this component is affected by faults and how these faults lead to a change in behavior. The fault-injection model captures hardware faults (relating to items of the technical architecture) as well as software faults. To express faults in a modular way, we represent the fault-injection model with *deviation models* as introduced in Section 4.1.5. We describe the extended logical architecture in Section 5.3.

Environment Specification An environment specification describes the context of a system, for example, its neighboring systems, its physical environment, or its users. Its purpose is again, to allow for a realistic availability analyses, as a system may show a different availability depending on its use. In our case, this artifact only consists of an *environment component model* describing the structure and behavior of the environment. An important constraint is that the interface of the environment needs to match the system interface. We discuss the environment specification in Section 5.4.

A central use case for our models are availability analyses, most importantly to verify the availability requirements. As shown by the *fulfills* relationship in Figure 5.1, the models of the artifacts availability specification, extended logical architecture and environment specification are used in combination to verify that the availability requirements are fulfilled. In Section 5.6 we describe this analysis in detail.

5.2 Availability Specification

The availability specification defines what availability should mean for the system under consideration. Especially, it provides a failure definition and specifies system-specific calculation rules for availability metrics. The availability specification has additional models to allow for a stepwise formalization, starting from informal models (the *failure mode list*) and going towards formal models (the *failure definition model*). This artifact consists of four models:

- *Failure Mode List*, which captures the relevant failure modes informally.
- *Failure Definition Model*, which formalizes the information from the failure mode list and furthermore describes which behavior of the system should be considered as a failure of which of the failure modes.
- *Failure Aggregation Model*, an optional model, which aggregates and simplifies the failure modes captured in the failure definition model. It thus facilitates the definition of availability metrics.
- *Availability Metric Model*, which provides formal calculation rules for availability metrics based on the models above.

5.2.1 Failure Mode List

Recall from Section 4.2.1 that when a failure occurs we can categorize it according to its *failure mode* and *severity level*. A failure mode is a certain type of failure (e.g. a signal is stuck at

	Failure Modes		**Severity Levels**	
Function	**Name**	**Description**	**Name**	**Description**
F1	FM 1	Informal description of FM 1	SL 1_1	Informal description of SL 1_1
			SL 1_2	Description of SL 1_2
	FM 2	Informal description of FM 2	SL 2_1	Informal description of SL 2_1
			SL 2_2	Description of SL 2_2
...				

Table 5.1: Schematic structure of a failure mode list showing two failure modes of a function, each having two levels of severity.

a certain value). The severity level specifies how serious the failure is (e.g. how important the stuck signal is). The *failure mode list* is a structured but informal description of failure modes relevant for the system. The failure modes in the list are grouped according to the system functions and the names of the system functions are annotated. For each failure mode, the failure mode list includes a description and a list of *severity levels* of this particular failure mode, again each with a description. Table 5.1 shows the schematic structure of a failure mode list. A formalized definition of the meaning of the failure modes is provided by the *failure definition model*, introduced below.

5.2.2 Failure Definition Model

The failure definition model describes which observed behavior at the system's interface is considered a failure of which failure mode. It consists of several *failure definitions*. A failure definition is a function that specifies, for a given history representing a part of the system's observable behavior, and for every point in time, if a failure of a specific failure mode is present and with which level of severity. A failure definition always references a subset of failure modes from the failure mode list (usually all failure modes for a given function).

We represent a failure definition by a *behavior function*. Let F be a set of channels, each channel representing one failure mode that should be considered by the failure definition. The type $\mathsf{type}(fm)$ of each channel $fm \in F$ specifies the severity levels of the failure mode represented by fm. If $I \triangleright O$ is the interface of the system or a particular system function, then a failure definition is modeled by a behavior function $D \in [I \cup O \triangleright F]$.

Figure 5.2 shows the example of a failure definition model and its relation to the syntactic interface of the considered system or function. It also illustrates the relationship to the failure modes and severity levels outlined in the failure mode list. In order to describe the behavior we can use any of the description techniques outlined in Section 4.1.3 (e.g. state-machines). We can also describe the behavior as a composition of several behavior functions.

Note, that we model a failure definition in the same way as we model, for example, a component and therefore use the same terminology (channels, histories). However, in order to avoid ambiguities and to stress the modeling intent, we will use the terms *failure mode channel* and *failure history* when we refer to channels and histories in the context of a failure definition.

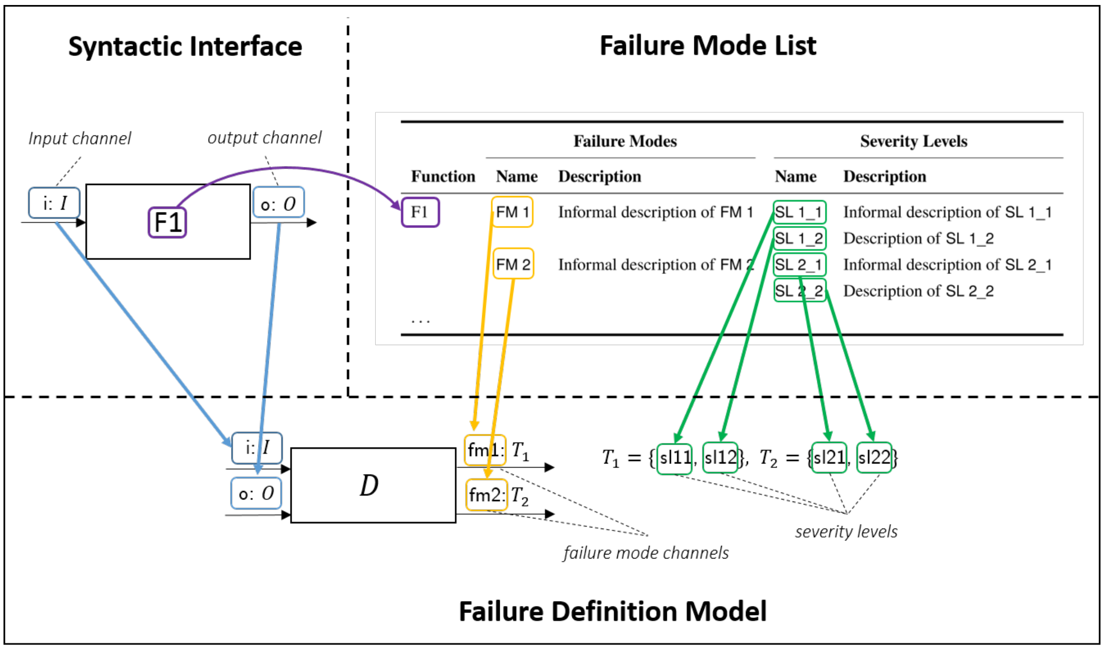


Figure 5.2: Relationship between a failure definition the syntactic interface of the considered system or system function, and the failure mode list.

5.2.3 Failure Aggregation Model

The failure mode list and the failure definition model provide a fine-grained description of failure modes and severity levels. In order to calculate concrete availability metrics, the information obtained from these models often needs to be aggregated. For example, the availability metric uptime can be easily calculated when the number of failure modes and number of severity levels are reduced to one (to denote “system is down”). We describe such an aggregation in a separate model, the *failure aggregation model*. Intuitively it serves as a mediator between the failure definition model and the availability metrics model.

Note, that a detailed, fine-grained description of failure modes in the failure mode list and failure definition model is still useful as a first step. The reason for this is that different aggregations may be necessary for different metrics and for different types of analyses. The original, fine-grained failure representation provides the common basis for those.

The failure aggregation model consists of several *aggregators*. An aggregator takes failures as inputs and calculates aggregated failures based on them. We represent an aggregator as a behavior function. As input channels an aggregator has a number of (fine grained) failure mode channels, as defined in the failure definition model. As output channels it has new failure mode channels, denoting the aggregated failure modes. The behavior of an aggregator can again be described using description techniques as introduced in Section 4.1.3 and as a composition of several behavior functions. Figure 5.3 illustrates the relationship between the aggregation model and the failure definition model.

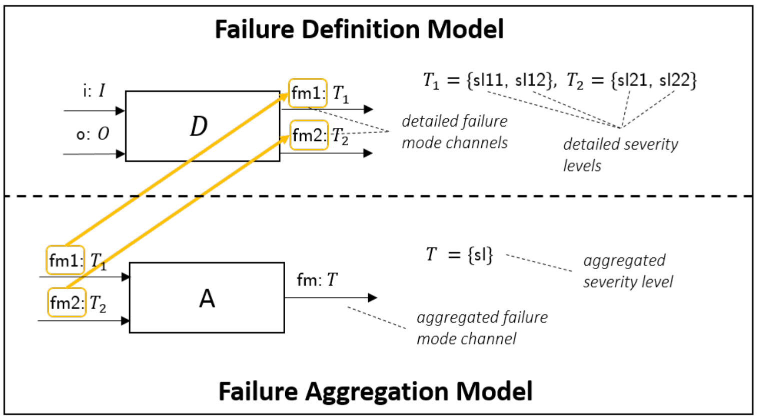

Failure Definition Model
i: I
o: O
D
fm1: T_1
fm2: T_2
T_1 = {sl11, sl12}, T_2 = {sl21, sl22}
detailed failure mode channels
detailed severity levels
fm1: T_1
fm2: T_2
A
fm: T
T = {sl}
aggregated severity level
aggregated failure mode channel
Failure Aggregation Model

Figure 5.3: Relationship between the failure definition model and the failure aggregation model. The failure mode channels serving as output of a failure definition are the input of an aggregator. The aggregator typically reduces the number of failure mode channels and severity levels.

5.2.4 Availability Metric Model

In the literature, there are numerous definitions of different types of availability metrics ranging from simple metrics, such as uptime, to more complex metrics with non-trivial calculation rules. Furthermore, availability metrics are system and project specific, depending on the failure modes and severity levels that are present in the failure definition. To capture these different availability metrics we use an explicit *availability metric model*. It captures calculation rules for concrete availability metrics.

To model availability metrics, we use functions $M : \overline{F}^\infty \to \overline{\mathbb{R}}$ mapping failure histories to a number, representing the value of the availability metric. Note that such a function has a different signature compared to a behavior function. In analogy to the term behavior function and as they are used to describe metrics, we use the term *metric function* for such functions. As they are not part of the formal theory presented in Chapter 4, we briefly develop the theory in the following sections.

Metric Functions

We start with the formal definition of metric functions, independent of its use for availability metrics.

Definition 25 (Metric Function)**.** A *metric function* with respect to a set of channels C is a measurable function $M : \overline{C}^\infty \to \overline{\mathbb{R}}$ that maps histories to a value from the extended reals. The set of metric functions for channel set C is denoted by $\mathbb{M}[C]$.

As we demand measurability, not every function mapping histories to a value qualifies as a metric function. Hence, when we define a metric function we have to verify that the underlying function is indeed measurable. To show measurability of a function, we have to show that the preimage of every measurable set is measurable. Although most functions that we intuitively come up with will be measurable, measurability is often not immediately obvious and may be tedious to prove. Therefore, we look for a more intuitive property of a quantitative specification that entails measurability. In Chapter 4.1.1 we showed that continuity qualifies as such a property in our setting.

Continuity. From Corollary 2 in Section 4.1.1 we know that continuity of a metric function entails measurability. Additionally, continuity of a metric function is easier to show than measurability. By Definition 15, a metric function M is continuous, if, for every x and for every $\epsilon > 0$, there exists a $\delta > 0$ such that $d_R(Q(x), Q(x')) < \epsilon$ for all x' with $d_C(x, x') < \delta$.

In our context, when a metric function is continuous the result for some infinite stream can be approximated by a different infinite stream with a sufficiently large (but finite) common prefix. In other words, continuity ensures that the result of metric functions when applied to an infinite stream can be approximated with any precision by looking at a finite prefix of this stream, if it is just long enough.

Continuity can often be shown more easily, compared to measurability. A typical hint to the continuity of a function is when the impact of later messages on the result diminish compared to earlier messages.

Time-Boundedness. Often, we are only interested in the first k time units for the definition of a metric function and the behavior afterwards is not relevant for the calculation. We call such metric functions time-bounded.

Definition 26 (Time-bounded)**.** A metric function M is called *time-bounded* if there exists $t \in \mathbb{N}_0$ such that for all histories z, z' with $z{\downarrow_t} = z'{\downarrow_t}$ it holds that $M(z) = M(z')$.

Theorem 6. *A time-bounded metric function is continuous.*

Proof. Let M be a time-bounded metric function. Given any history h and some $\epsilon > 0$, there exists $t \in \mathbb{N}_0$ such that $M(h) = M(h')$ for all h' with $h{\downarrow_t} = h'{\downarrow_t}$. We choose $\delta = 2^{-t}$. For any h' with $d_C(h, h') < \delta$, it holds that $h{\downarrow_t} = h'{\downarrow_t}$, due to the definition of d_C. Therefore, $M(h) = M(h')$ due to the time-boundedness of M and hence $d_R(M(h), M(h')) = 0 < \epsilon$. Therefore, M is continuous. □

Weighted Automata

A way to represent a metric function is via a state transition model. Similar to the probabilistic automata introduced in Section 4.1.3, we also attach quantities to the transitions. The general idea is then to aggregate the values along an execution path according to some aggregation function in order to obtain a single value. If the number of states is finite, such a state machine is called a *weighted finite automaton* (WFA). A formal definition of WFA, taken from Chatterjee et al. (2009), is presented below.

Definition 27 (Weighted Finite Automaton)**.** A *weighted finite automaton* (WFA) for channel set C is represented by a tuple(S, S_0, δ, w) where

- S is the state space,
- $S_0 \subseteq S$ is a set of initial states,
- $\delta \subseteq S \times \overline{C} \times S$ is the transition relation labeled with channel valuations,
- $w : \delta \to \mathbb{Q}$ is a weight function that assigns a weight, given as a rational number, to each transition of the automata.

An infinite *run* of a WFA $\mathcal{A}$ is a sequence $r = ((s_0, v_0, s_1), (s_1, v_1, s_2), \ldots)$, such that $s_0 \in S_0$ and $(s_i, v_i, s_{i+1}) \in \delta$, for all i. We write $\mathit{runs}(\mathcal{A})$ for all runs of $\mathcal{A}$. With $\mathit{hist}(r)$ we denote the history embedded inside the run r.

$$\mathit{hist}((s_0, v_0, s_1), (s_1, v_1, s_2), \ldots) = \langle v_0, v_1, \ldots \rangle,$$

With $\mathit{runs}_{\mathcal{A}}(h)$, we denote the set of runs that produce h.

$$\mathit{runs}_{\mathcal{A}}(h) = \{r \in \mathit{runs}(\mathcal{A}) \, : \, \mathit{hist}(r) = h\} \ .$$

For a run r, $\mathit{weights}_{\mathcal{A}}(r)$ is the sequence of weights associated with r.

$$\mathit{weights}_{\mathcal{A}}(r).i = w(r.i)$$

Given a value function $V : \mathbb{Q}^{\infty} \to \mathbb{R}$ mapping infinite sequences of rational numbers to a real number, a WFA $\mathcal{A}$ induces the metric function $Q_{\mathcal{A}}$ given by

$$Q_{\mathcal{A}}(h) = \sup \{ \, V(\mathit{weights}_{\mathcal{A}}(r)) \, : \, r \in \mathit{runs}_{\mathcal{A}}(h) \} \ .$$

Note that a history can be produced by multiple runs, for instance due to non-determinism in the automaton. From the possibly many weighting results we choose a distinct one by selecting the supremum. Examples for value functions V for weighted automata are (Chatterjee et al., 2009):

- $\mathit{LimInf}(v) = \liminf_{n \to \infty} v.n$
- $\mathit{LimAvg}(v) = \liminf_{n \to \infty} \frac{1}{n} \sum_{i=0}^{n-1} v.i$

These functions are well-defined as long as there is a lower bound on the values in v.

Description Techniques

Metric functions can be described in various ways. Here, we consider two description techniques: using logical terms and using weighted state transition diagrams. In any case, we embed descriptions of metric functions into a frame with frame label MF and state the involved channel names and types.

M	MF
in $c_1 : T_1, \ldots, c_k : T_k$	
body	

Specification with Logical Terms. Similar to specifying behavior with logical formulas we can define a metric function by giving a logical term that evaluates to a real number and that has the channel names as free variables. A specification for channels $C = \{\mathsf{c}_1, \ldots, \mathsf{c}_k\}$ is given by a term t with variables $c_1, \ldots, c_k$. This term induces a metric function $M \in \mathbb{M}[C]$ defined as

$$\forall c \in \overline{C}^{\infty} : M(c) = t[c_1 \mapsto c|_{\mathsf{c}_1}, \ldots, c_k \mapsto c|_{\mathsf{c}_k}]$$

Specification with State Transition Diagrams. State transition diagrams for weighted automata resemble transition diagrams for I/O automata introduced before. A transition is annotated with a pre-condition, an input-binding and a post-condition. Additionally, a weight in braces is added to a transition. Figure 5.4 illustrates a transition diagram for a weighted automaton. In the illustration, the top transition carries weight w_1 and the bottom transition carries weight w_2. Additionally, we add a valuation function to the specification frame, labeling it with the keyword **val**. The denotation of such a specification is the metric function induced by the WFA described in the diagram.

WFA ═══ MF

in $c_1 : T_1, \ldots, c_k : T_k$
local $l_1 : T_{l1}, \ldots, l_n : T_{ln}$
val V

$$\begin{tikzcd}
\text{start} \arrow[r] & \boxed{S_1} \arrow[r, bend left, "\{\ldots\}\ldots\{\ldots\} \; (w_1)"] & \boxed{S_2} \arrow[l, bend left, "\{\ldots\}\ldots\{\ldots\} \; (w_2)"]
\end{tikzcd}$$

Figure 5.4: Schematic structure of a quantitative specification described using a weighted state transition diagram. The transitions of the diagram are additionally annotated with weights (e.g. w_1, w_2 in the diagram). Furthermore the specification is extended by a valuation function V.

Relationships to other Models

As indicated by Figure 5.1, the availability metrics model has a relationship to the failure aggregation model and to the failure definition model. This relationship is illustrated in Figure 5.5. The figure shows that the availability metrics model references the failure mode channels of both, the failure aggregation model and the failure definition model. These failure mode channels form the inputs to the metric functions specified in the availability metrics model.

5.3 Extended Logical Architecture

The *extended logical architecture* extends the original logical architecture such that it includes the behavior in the case of faults. As faults can lead to a situation where the system behavior deviates from the specification (i.e. failures), faults and their effect on the system behavior need to be considered in an availability analysis. Capturing faults and their effects is the purpose of the *fault-injection model*, which is the only model in this artifact.

With the *fault-injection model* we capture the effects of hardware and software faults, by describing behavior deviations of the components in the logical architecture. We consequently represent the fault-injection model with *deviation models*, introduced in Section 4.1.5. That

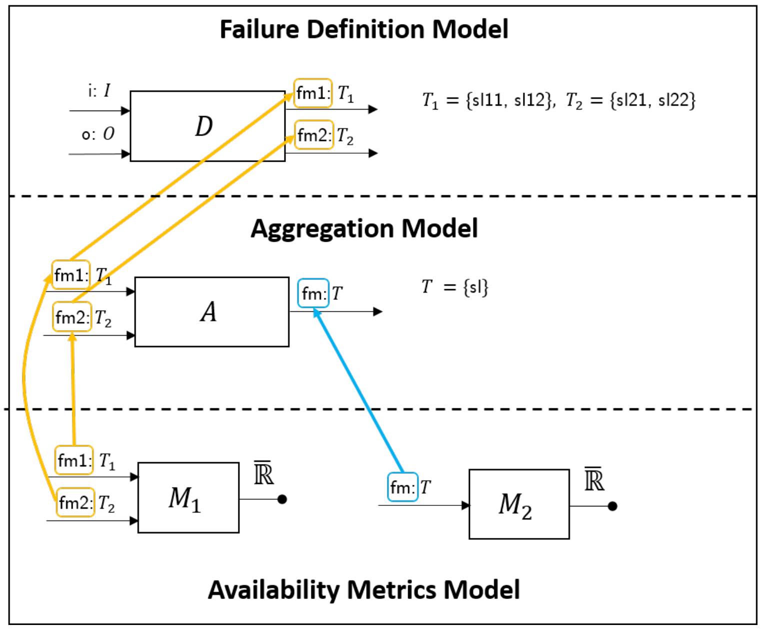


Figure 5.5: Relationship between the availability metrics model, the failure aggregation model and the failure definition model. The availability metrics model may refer either to the failure mode channels of an aggregator (M_2) or directly to the failure mode channels of a failure definition (M_1).

means, for a component in the logical architecture with interface $I \triangleright O$ the fault-injection model may contain one or more deviation models $D \in \mathbb{D}[I \triangleright O]$ applied to this component.

Figure 5.6 illustrates the relationship between the logical architecture and the extended logical architecture with the applied fault-injection model.

5.4 Environment Specification

Statements about the availability of a system are only valid for a certain environment. Depending on the behavior of external systems or users, the system under development is likely to show different availability characteristics. For instance, if a certain device has a higher failure rate under high workload than under low workload, the availability of every function that involves this device depends on the workload on this device. Therefore, any availability analysis has to consider the behavior of the environment. For this reason we incorporate the environment into our artifact model by introducing an explicit environment specification.

In our artifact model, the environment specification only consists of one model: We employ the *environment component model* to specify the interface, the internal structure and the behavior of the environment. This is a valid approach because, from the point of view of the system under development, the environment is again a system (or many systems). Therefore, we can model the environment in the same way as we model a system. The syntactic interface of an environment needs to match the system's interface. This means, it needs to provide the inputs to the systems and may process the system's outputs. Semantically, the environment model is captured by (a composition of) probabilistic behavior functions. Therefore, we can

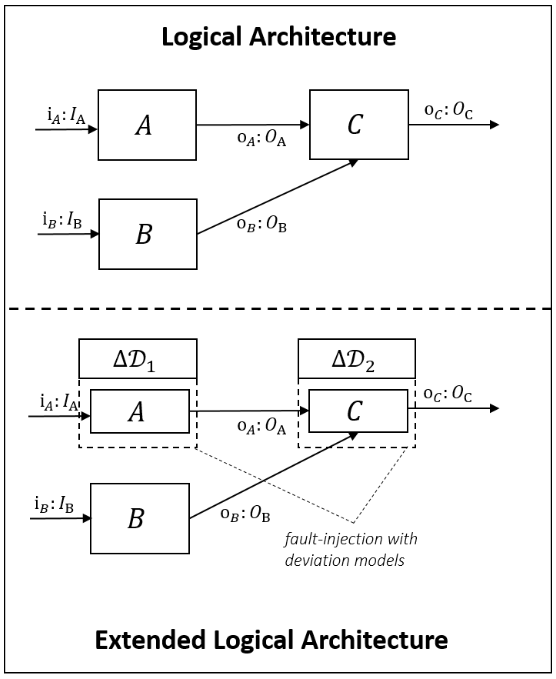


Figure 5.6: Relationship between the logical architecture and the extended logical architecture.

use the same description techniques, such as logical formulas and tables to describe black-box behavior, state transition diagrams to describe state behavior and data flow networks to describe the decomposition of an environment into multiple environment components. Below, we state the formal definition of an environment.

Definition 28 (Environment)**.** An *environment* of a system $S \in [I \triangleright O]$ is a behavior function $E_S \in [O \triangleright I]$. By $Env(S)$ we denote the set of all environments of S.

Note that the environment is not only providing inputs to the system but is also reacting on the system's outputs. As shown by Broy (Broy, 1998) this is necessary, as in the case of non-deterministic systems otherwise not all environment assumptions can be represented. Note further that for every system $S \in [I \triangleright O]$ and environment $E_S \in Env(S)$, the composition $S \otimes E$ is a function without inputs and with $I \cup O$ as outputs, hence $S \otimes E \in [\varnothing \triangleright I \cup O]$. The result of the execution $\langle\!\langle S \otimes E \rangle\!\rangle$ is a set of probability measures on I/O histories of the system, i.e. $\langle\!\langle S \otimes E \rangle\!\rangle \in \mathbf{Pr}(\overline{I \cup O}^{\infty})$.

Figure 5.7 illustrates the relationship between the environment specification and the extended logical architecture with an example. The inputs to the system match the outputs of the environment. On the other hand, the outputs of the system match the inputs to the environment.

5.5 Availability Requirements Specification

Availability requirements describe the demands of the stakeholders with respect to the system's availability properties. We consider two types of models of this artifact: *textual availability descriptions*, which capture availability requirements informally by using natural language text,

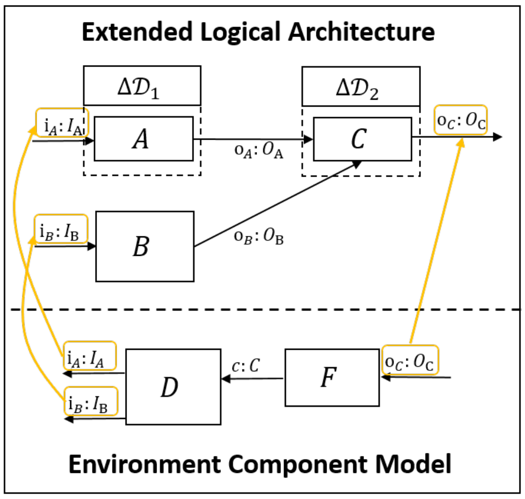


Figure 5.7: Relationship between the environment model and the logical architecture. The inputs to the system match the outputs of the environment and vice versa.

and *availability constraints*, which are formal statements referring to availability metrics defined in the availability specification.

5.5.1 Textual Availability Descriptions

Textual availability descriptions capture availability requirements informally by natural language text. These informal, textual descriptions refer to the availability of the system, single functions or function groups. They may refer to availability metrics or contain qualitative statements about the desired availability. Especially in early development phases, demands on availability are often still vague and hard to formalize. In this situation, textual descriptions provide a means to document these initial requirements. In a second step they can be formalized by writing formal *availability constraints*.

5.5.2 Availability Constraints Model

Availability constraints formalize, refine and extend the textual availability descriptions. Availability constraints specify precisely which ranges should be allowed for which availability metric. The metrics to which an availability constraint refers to are defined in the availability metrics model (Section 5.2.4), which is part of the availability specification. We represent availability constraints by a pair, consisting of a metric function and a value range. The formal definition is given below.

Definition 29 (Availability Constraint)**.** An *availability constraint* is a pair (M, R), where $M \in \mathbb{M}[F]$ is a metric function for some set of failure modes F, and $R \subseteq \overline{\mathbb{R}}$ is a subset of the reals that describes the range of values that should be allowed for the given metric.

Note, that the metric function that is part of an availability constraint stems from the availability metric model (Section 5.2.4). By $\mathbb{C}$, we denote the set of availability constraints. The availability constraints model has two relationships to other models. On the one hand, it formalizes textual availability descriptions and on the other hand it references availability metrics from the availability metrics model. These relationships are illustrated in Figure 5.8

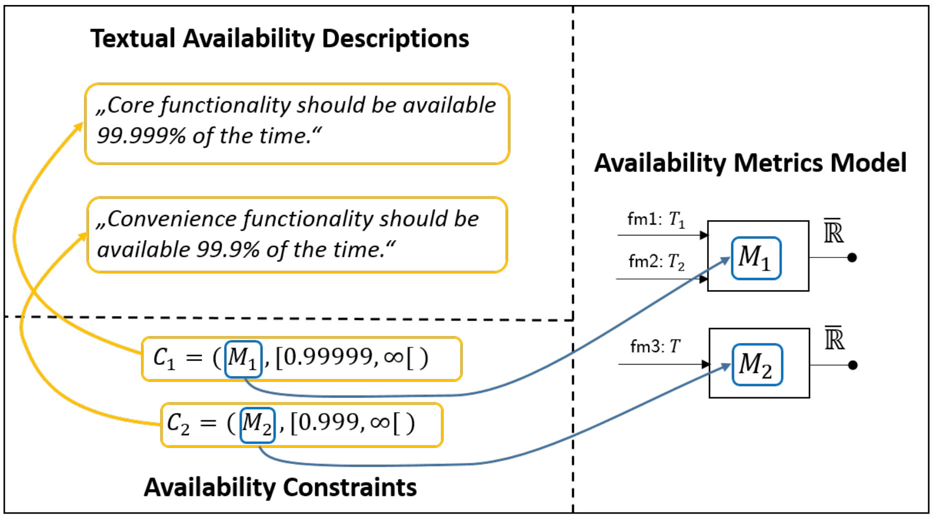

Textual Availability Descriptions
„Core functionality should be available 99.999% of the time."
„Convenience functionality should be available 99.9% of the time."
$C_1 = (M_1, [0.99999, \infty[)$
$C_2 = (M_2, [0.999, \infty[)$
Availability Constraints
Availability Metrics Model
fm1: T_1
fm2: T_2
fm3: T
M_1
M_2
$\overline{\mathbb{R}}$
$\overline{\mathbb{R}}$

Figure 5.8: Relationship between the textual availability description, availability constraints and the availability metrics model. The availability constraints formalize the textual availability requirements. For that, it references the formally defined availability metrics.

5.6 Availability Analysis

In this section, we will not introduce further modeling artifacts. Instead, we show how to use the artifacts presented in the preceding sections to verify availability requirements.

5.6.1 Prerequisites

The starting point for the analysis are the following artifacts and models.

Extended Logical Architecture The extended logical architecture captures the interface, the structure and the behavior of the system. The extended logical architecture includes the components model from the original logical architecture extended by the fault-injection model. For the analysis we are only interested in the overall behavior at the system interface. Therefore we represent the whole extended logical architecture with a single behavior function $S \in [I \triangleright O]$.

Environment Specification According to our artifact model we capture the system environment by the environment component model. Similar to the extended logical architecture we are also only interested in the overall behavior of the environment. Hence, we also represent the environment specification by a single behavior function $E \in Env(S)$.

Availability Specification From the availability specification we use failure definition model and the failure aggregation model.

- *Failure Definition Model* The failure definition model consists of several failure definitions, all represented by behavior functions. As we can compose these behavior functions, we assume a single behavior function $D \in [C \rhd F_D]$ where $C \subseteq I \cup O$ is the subset of the syntactic interface of the system that the failure definitions consider and F_D is the set of detailed failure mode channels.
- *Failure Aggregation Model* The failure aggregation model consists of a number of aggregators that we again represent together as behavior function $A \in [F'_D \rhd F_A]$, where $F'_D \subseteq F_D$ is the subset of failure mode channels subject to aggregation and F_A is a set of aggregated failure mode channels.

Composing the above behavior function and restricting the result to the failure mode channels $F = (F_D \cup F_A)$, yields

$$\mathcal{M} = (S \otimes E \otimes D \otimes A)\dagger F \ .$$

This composition is schematically illustrated in Figure 5.9. Note that the syntactic interface of the obtained composition is $(\varnothing \rhd F)$. The execution of it yields a set of probability measures over the failure mode channels. This means, formally

$$\langle\!\langle \mathcal{M} \rangle\!\rangle \subseteq \mathbf{Pr}(\overline{F}^{\infty}) \ .$$

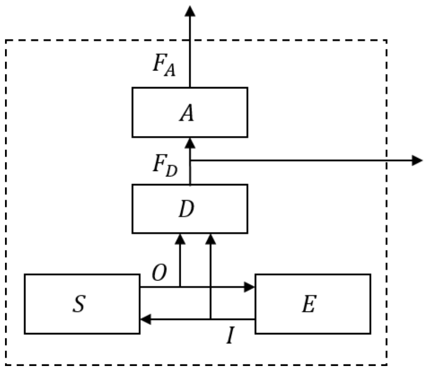


Figure 5.9: Schematic illustration of the composition of the prerequisite models.

5.6.2 Availability Verification and Measurement

Our semantic model of availability metrics are metric functions, as outlined in Section 5.2.4. By definition, a metric function is a measurable function and can therefore act as random variable. Hence, if we have a suitable probability measure, we can obtain the expected value of a metric function.

At this point, we include the last artifact of our artifact model, the *availability requirements specification*. More specifically we consider the availability constraints model. We represented this model by a set of availability constraints of the form (M, R). Based on this, we formalize the fulfillment of availability requirements by demanding that the expected value of a metric function M is in the allowed range given by R. We assume for the following definition a set of failure mode channels F and an indexed set of availability constraints $\mathcal{C} \subset \mathbb{C}$, such that $\forall (M_i, R_i) \in \mathcal{C} : M_i \in \mathbb{M}[F_i]$ with $F_i \subseteq F$.

Definition 30 (Fulfillment of Constraints)**.**

- Given a probability distribution $\mu \in \mathbf{Pr}(\overline{F}^{\infty})$. We say that μ fulfills $\mathcal{C}$ and write $\mu \vDash \mathcal{C}$, if $\forall (M_i, R_i) \in \mathcal{C} : \mathbf{E}_{\mu \dagger F_i}[M_i] \in R_i$.

- Given a set of probability distributions $\Theta \subset \mathbf{Pr}(\overline{F}^{\infty})$. We say that Θ fulfills $\mathcal{C}$ and write $\Theta \vDash \mathcal{C}$, if $\forall \mu \in \Theta : \mu \vDash C$.

With the above definitions, we formalize the fulfillment of availability requirements: Let $\mathcal{M}$ be the composition of the relevant availability models as defined in Section 5.6.1 with $\langle\!\langle \mathcal{M} \rangle\!\rangle \subset \mathbf{Pr}(\overline{F}^{\infty})$. The fulfillment of the availability requirements, formalized as a set of availability constraints $\mathcal{C} \subseteq \mathbb{C}$ can then be formulated as

$$\langle\!\langle \mathcal{M} \rangle\!\rangle \vDash \mathcal{C} \ .$$

Often we do not only want to know whether the requirements are fulfilled, but are interested in the actual values of the availability metrics (possibly with certain frame conditions). Such an information is, for instance, necessary to perform a sensitivity analysis for identifying the impact of system parameters on the availability. In the case of non-deterministic systems, the result for such a query is not a single value but is a range of availability metrics. To single out a value we choose the supremum or infimum of this range. Which one, depends on the situation and the meaning behind a metric.

Definition 31 (Expected Value under Constraints)**.** Given an availability metric $M \in \mathbb{M}[F_M]$, a set of probability measures $\Theta \subset \mathbf{Pr}(\overline{F}^{\infty})$ with $F_M \subseteq F$, and a set of constraints $\mathcal{C} \subset \mathbb{C}$. The maximal (minimal) *expected value* of M obtained from probability measure Θ, while fulfilling constraints $\mathcal{C}$, is the supremum (infimum) of the expected values of M for all measures that fulfill the constraints. Formally,

$$\begin{aligned} \mathit{Exp}^*(M, \mathcal{C}, \Theta) &= \sup \{\mathbf{E}_{\mu \dagger F_M}[M] \, : \, \mu \in \Theta \wedge \mu \vDash \mathcal{C}\} \ , \\ \mathit{Exp}_*(M, \mathcal{C}, \Theta) &= \inf \{\mathbf{E}_{\mu \dagger F_M}[M] \, : \, \mu \in \Theta \wedge \mu \vDash \mathcal{C}\} \ . \end{aligned}$$

Applying this definition to our artifact model, we can query the maximal attainable value for an availability metric M while simultaneously satisfying constraints $\mathcal{C}$ on other metrics, by

$$\mathit{Exp}^*(M, \mathcal{C}, \langle\!\langle \mathcal{M} \rangle\!\rangle) \ .$$

5.7 Application to Related Concepts

In Section 4.2.3, we distinguished availability from several related concepts, among them reliability and safety. In the following section, we provide a more formal discussion of the relationship between these terms. We base this discussion on a quantitative notion of correctness. We show that the artifact model, outlined in this chapter, can be understood as a means to describe quantitative correctness. Based on this, we sketch how the artifact model for availability can also be applied to reliability and safety.

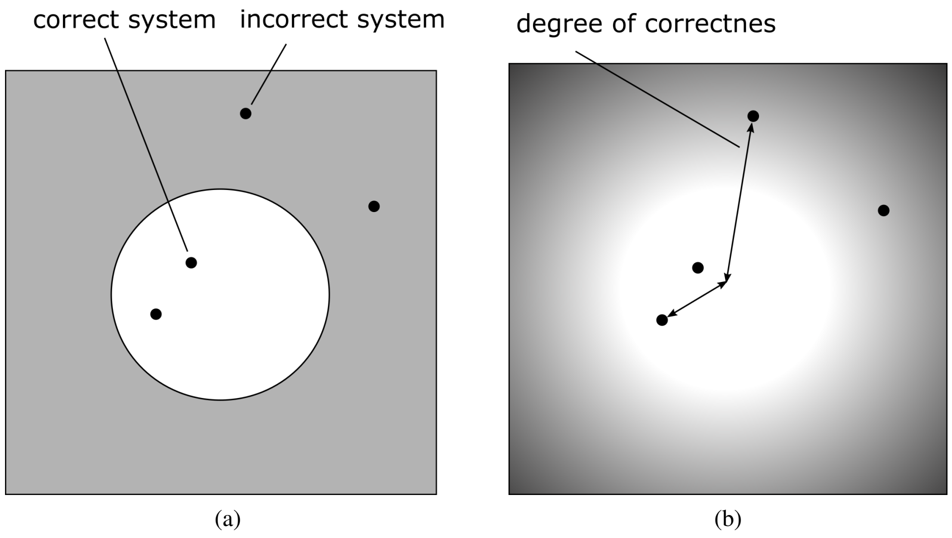


Figure 5.10: Illustrations of (a) the boolean notion of correctness and (b) the quantitative notion of correctness. In the quantitative notion, the crisp specification, visualized by the circle, is replaced by a quantitative degree of correctness, visualized by a color gradient.

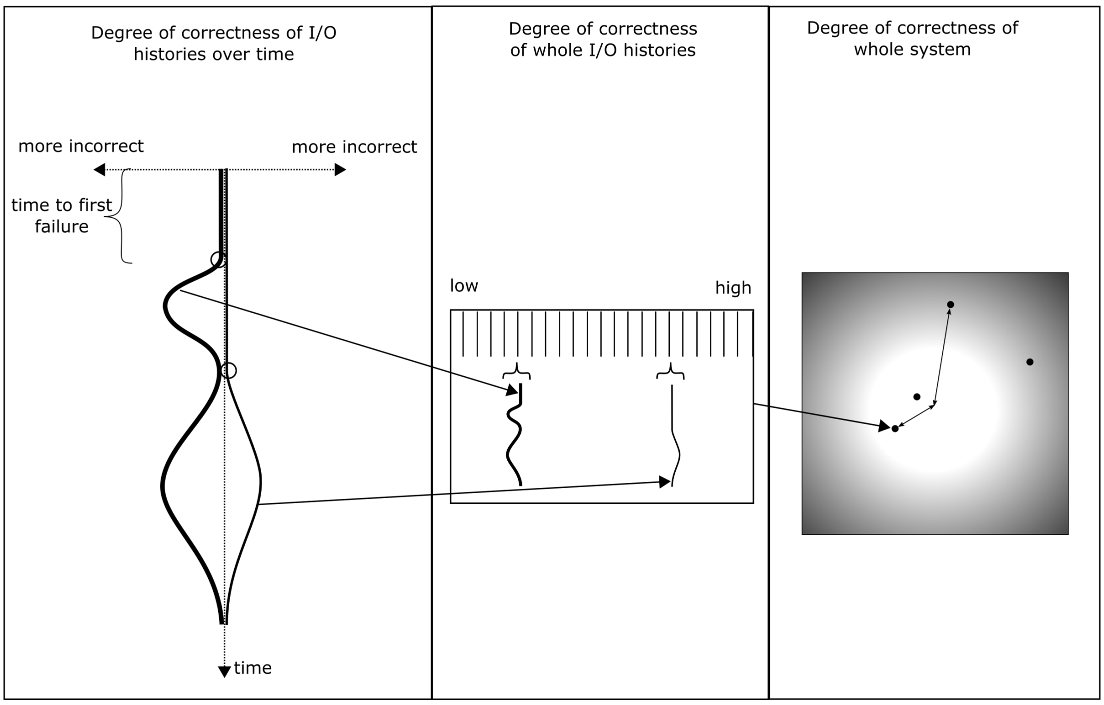

Degree of correctness of I/O histories over time
Degree of correctness of whole I/O histories
Degree of correctness of whole system
more incorrect
more incorrect
time to first failure
low
high
time

Figure 5.11: Illustration of the degree of correctness on different levels of aggregation. The figure on the left illustrates the degree of correctness of individual histories over time. The illustration in the middle visualizes the aggregated degree of correctness of whole histories. Finally, the right figure visualizes the degree of correctness of the whole system as an aggregation of the degrees of correctness of its histories.

5.7.1 Correctness

We call a system correct if it behaves according to its specification. The traditional notion of correctness is binary: a system is either correct or incorrect, but nothing in between. The binary notion of correctness is illustrated in Figure 5.10a. In reality, a more differentiated view on the fitness of a system is often desirable. Not all systems that are incorrect in the boolean sense are equally undesired. On the other hand, of all correct systems some may be preferred over others (Henzinger, 2010).

To accommodate nuances in the evaluation of a system, quantitative notions of correctness can be established. From such a point of view, a system is not incorrect, but correct to a certain (possibly low) degree. This degree can be thought of as distance between the system and its specification. See Figure 5.10b for an illustration of this idea. Availability metrics are an example of such a quantitative notion of correctness. The value of an availability metric can be interpreted as a degree of correctness.

We can apply the transition from a boolean notion of correctness to a quantitative notion of correctness not only to whole systems, but also to single input/output histories. Thereby we can derive the degree of correctness of the whole system from the degrees of correctness of its input/output histories. This situation is illustrated in Figure 5.11. The illustration consists of three connected parts. Each part illustrates the degree of correctness on a different level of aggregation. The left part shows the degree of correctness of two histories over time. The y-axis denotes the time and the x-axis the degree of correctness of the history at this point in time. The histories are illustrated by the curves leading from the top to the bottom. The larger the displacement of a curve at some point, the lower is the degree of correctness at this point. In a probabilistic setting, input/output histories are associated with a probability[1], describing how likely it is to observe this history. In the illustration, we map this probability to the thickness of a curve. For example, the left curve has a higher probability than the right one. That means, it is more likely to observe the history represented by the left curve, than the one represented by the right curve. The middle part of Figure 5.11 shows the aggregated degree of correctness of whole histories. Each history is assigned a degree of correctness, visualized by the horizontal position in the diagram, where a position to the left denotes a low degree of correctness and a position on the right denotes a high degree of correctness. Finally, the right part of the figure denotes the further aggregated degree of correctness for the whole system, which is derived from the degrees of correctness of the individual histories.

We can interpret availability as a form of quantitative correctness. This point of view opens a new perspective on our artifact model: it can be explained in terms of quantitative correctness in the following way: The failure definition model provides means to define measurements for the degree of correctness of a history over time (the displacements of the behavior curves in the left part of Figure 5.11). That means, without a failure definition model, it would not be possible to draw a curve, as the displacement could not be determined. The fault injection model provides realistic histories in the presence of faults. In the example, this is illustrated by the curves moving to the left or right. Without the fault injection model, we would only obtain straight lines. The availability metric model aggregates the degrees of correctness for all points of an input/output history to a single value, denoting the degree of correctness (the availability) associated with a history. This relates to the transition from the left part of Figure 5.11 to the middle part. Typically, an availability metric will relate to the overall displacement of a curve, visualized by the area between the curve and the middle line in Figure 5.11. The transition to

[1]This is a simplification, as probilities, in general, cannot be attached to individual trajectories.

a system-wide degree of correcteness (i.e. the transition to the right part of Figure 5.11) is, in our case, done by obtaining the expected value of the availability over all input/output histories.

5.7.2 Relation to Other Dependability Properties

We saw in Section 4.2.3 that there are concepts closely related to availability. Most of these concepts have in common that they also constitute a form of quantitative correctness. In the following, we discuss two of them, reliability and safety, in the terms of quantitative correctness and relate them to our artifact model. By this, we demonstrate that our artifact model is potentially applicable to more concepts than availability alone. Furthermore, we can carve out the differences and commonalities between the concepts in more detail.

Reliability

The first property we study in this context, is reliability. Similar to availability, we can define reliability in terms of our model of quantitative correctness. Reliability relates to the time of the first failure. From the viewpoint of reliability, histories that represent a failure-free behavior have a higher degree of correctness than those that exhibit a failure early. In the left part of Figure 5.11, we highlighted the point where a curve leaves the middle line (and thus, exhibits a failure) for the first time with a circle. Note, that the right of the two curves in the illustration leaves the middle line later than the left curve. Hence, also from the viewpoint of reliability, the according history has a higher degree of correctness. Finally, we can transition to a system wide value for the degree of correctness by deriving the expected value over all histories.

Since we can charactarize reliability in terms of the same model of quantitative correctness as availability, we can also apply our artifact model to describe reliability: The failure definition model plays the same role as it does for availability: It measures the degree of correctness of a history at each point in time. This is visualized in Figure 5.11 by the displacement of the curves. The fault injection model again produces realistic histories. Formally, a metric model for reliability is equivalent to an availability metric model. The difference is that an availability metric model typically relates the degree of correctness of a history to the frequency, duration and severity of failure periods (visualized by the area between the curve and the middle line in the left part of Figure 5.11), while a reliability metric model typically relates the degree of correctness to the time until the first failure (visualized by the y-position of the circle in the left part of Figure 5.11). To obtain the system reliability, we again calculate the expected value of the reliability metric from the probability distribution over the I/O histories.

Safety

Safety can also be understood as a form of quantitative correctness: Often, a system is not either completely safe or unsafe, but safe to a certain degree. We describe the safety of the system by quantifying the risk that the system causes harm to its environment to certain extent. There are several metrics for the quantitative assessment of safety. Janicak (2009) discusses a number of such metrics. Often, safety is captured by the financial risk (e.g. payment of compensation, higher insurance costs, etc.) due to safety incidents.

In the context of Figure 5.11, the degree of correctness of an I/O history for any given point in time, from the viewpoint of safety, relates to the amount of harm that the system is causing to its environment at this point in time. The degree of correctness of a whole history then relates to the overall amount of harm that the system causes with this behavior. Finally, the degree of correctness of the overall system then relates to the expected value of the harm caused by a system during its operation. This fits to the model of quantitative correctness we introduced above.

As we can capture safety as a form of quantitative correctness, we can formally also apply our artifact model. However, there is a practical challenge in the case of safety: An important difference between the consideration of safety and availability is, that the dynamic of the environment plays a more important role for safety. The harm that a system causes is much harder to assess only by observing the system's behavior. Instead, we have to evaluate the effect of the system behavior in the environment. For instance, the failure of a train control system to apply the safety break when requested may cause harm to passengers if the train is currently in a train station, passengers are entering and exiting the train, and the train starts moving. Hence, it depends on the current state of the environment if harm is actually caused. Therefore, to apply our artifact model to safety, we have to accommodate the environment for the assessment of harmful incidents. This could be done by merging the failure definition model and the environment model to form an integrated *environment and failure model*. An alternative would be to expose information of the state of the environment at the environment interface and to use this information in the failure definition model.

Discussion

We saw that reliability and safety can be studied in the context of quantitative correctness and described using an artifact model similar to the one we employed for availability. This highlights the connection of safety and reliability models to other parts of a system description. In particular, it reveals their relationship to the functional requirements and the functional specifications of the system. This furthermore challenges the view that these properties are non-functional. Integrating these additional properties into one model enables obtaining a comprehensive description of the system with only few redundancies and makes it possible to analyze different quality attributes of the system in a consistent way.

5.8 Summary

In this chapter, we introduced modeling artifacts for availability modeling and analysis based on a formal modeling theory. These artifacts are:

- *Availability Specification*, to capture and define the relevant failure modes and availability metrics,
- *Extended Logical Architecture*, to extend the nominal logical architecture with behavior in case of faults,
- *Environment Specification*, to capture the behavior of external systems and users of a system,
- *Availability Requirements Specification*, to capture and formalize the availability related demands on the system.

We further showed how these models can be combined together to perform availability analyses. The analyses that we considered are the verification of availability requirements, given in the form of constraints on the metric values, and the derivation of concrete availability metric values.

Finally, we embedded our artifact model into a general model of quantitative correctness and discussed the properties reliability and safety from this point of view.

Chapter 6

Availability Modeling Method

6.1 Overview

In this chapter we complement the artifact model from Chapter 5 with a modeling method. The method provides a systematic way to develop concrete model instances. The first part of the method is a set of *basic building blocks*, which are reusable specifications from which modelers can assemble custom availability models. The second part of the method is a *modeling process*, describing a sequential path through the artifact model. Additionally, for some of the models presented in Chapter 5, we provide *step-by-step guides*, supporting the systematic creation of models, and *modeling patterns*, which provide a basic model structure that only needs to be filled in by a modeler.

The chapter is structured as follows: In Section 6.2 we first introduce a running example used to illustrate our approach throughout the chapter. We use the example of a data storage and access system, introduced by Broy (2010b). Afterwards, in Section 6.3, we present the basic building blocks for availability modeling. Finally, in Section 6.4, we present the modeling process and show how we support the creation of the individual models using basic building blocks, step-by-step guides and patterns, and demonstrate the method with our running example.

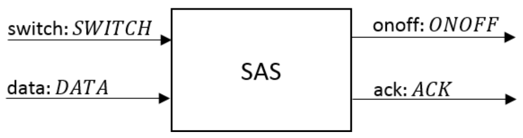
6.2 Running Example: Data Storage and Access System

To illustrate the approach outlined in this chapter, we use an example introduced by Broy (Broy, 2010b). The storage and access system (SAS) can store a number and reproduce the stored number on demand. Furthermore, the whole system can be switched off and on again. When it is switched off, it ignores requests for storing a new number or reproducing the stored number. However, it does not lose its stored number. In the following, we give the models for functional, logical and technical architecture that describe the SAS system.

6.2.1 Functional Architecture

The system is characterized by the syntactic interface depicted in Figure 6.1. The input channel switch is used for switching the system on and off. The input channel data is used to provide data or commands for outputting the stored number. The output channel onoff signals whether the system is currently on or off and the output channel ack acknowledges the receipt of a piece of data or a read command.

We follow Broy (2010b) and decompose the system into two system functions: Switch, for switching the system on and off and Access, providing the storage and accessing function. (Figure 6.2). Each system function relates to a sub-interface of the SAS System. The two functions interact via the common mode channel mode_onoff.



(a)

Type	Message Set
$SWITCH$	$\{\mathsf{sw}\}$
$DATA$	$\{\mathsf{read}\} \cup \{\mathsf{set}(n) : n \in \mathbb{N}\}$
$ONOFF$	$\{\mathsf{on}, \mathsf{off}\}$
ACK	$\{\mathsf{done}\} \cup \mathbb{N}$

(b)

Figure 6.1: Syntactic interface (a) and corresponding datatypes (b) of the SAS example system.

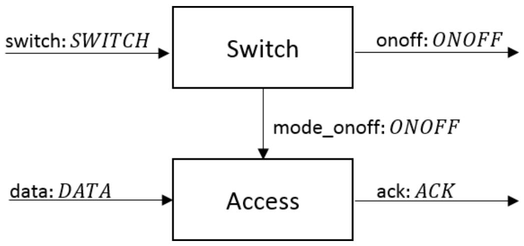


Figure 6.2: Functional architecture of the SAS example system. It consists of two system functions Switch and Access that interact via a common mode channel.

The Switch function toggles between on and off on reception of an sw signal. It propagates the current on/off state via the mode channel mode_onoff. When a switch happens, the new state is also output via the channel onoff. The second function, Access, distinguishes two situations based on mode_onoff. If the mode is off, it does not react on any stimulation via the data channel. If the mode is on and it receives $\mathsf{set}(k)$, it replaces the internally saved number with k and acknowledges the replacement by outputting the message done. It reproduces the internally stored number if it receives the message read. This behavior of the functions is formally described by the specifications given in Figure 6.3.

6.2.2 Logical Architecture

The logical architecture used to realize the SAS system is quite differently structured compared to the functional architecture. Nevertheless the interface behavior described by the logical architecture matches the interface behavior described by the functional architecture. We assume a fault-tolerant logical architecture consisting of two redundant stores, which are responsible for storing and reproducing the numbers, as well as a controller coordinating the stores and implementing the switching. Figure 6.4 shows the logical architecture of the SAS system on the highest hierarchy level. The interface between the controller and the two stores includes

Switch

in	switch : $SWITCH$
out	onoff : $ONOFF$
	mode_onoff : $ONOFF$
local	m : $ONOFF$ **initial** on
univ	s : $ONOFF$

Input		Output		
m	switch	m'	mode_onoff	onoff
s	□	s	s	□
off	sw	on	on	on
on	sw	off	off	off

Access

in	mode_onoff : $ONOFF$
	data : $DATA$
out	ack : ACK
local	v : $\mathbb{N}$ **initial** 0
univ	j : $\mathbb{N}$
	k : $\mathbb{N}$

Input			Output	
v	mode_onoff	data	v'	ack
j	off	?	j	□
j	on	read	j	j
j	on	set(k)	k	done
j	on	□	j	□

Figure 6.3: Specification of the Switch and Access system functions of the SAS example system.

the types *DATA_ID* and *ACK_ID*. These are extensions of the types *DATA* and *ACK* used in the functional architecture. Messages of these extended types are pairs (d, id) where the first part is a message of type *DATA* or *ACK* respectively. The second part is a message sequence number. The sequence number is used in the controller to identify responses of the stores to the same request. The store's behavior is similar to the Access function, however, it does not consider any on/off switching, as this is centrally handled in the controller. The controller consists of three components (see Figure 6.5). One of these components (SwitchC) is defined exactly as the Switch function. The other two components are responsible for distributing the received signals to the redundant stores (Store Forward) and merging the stores' responses (Store Merge). The specifications of all components of the logical architecture are given in Figures 6.5 and 6.6.

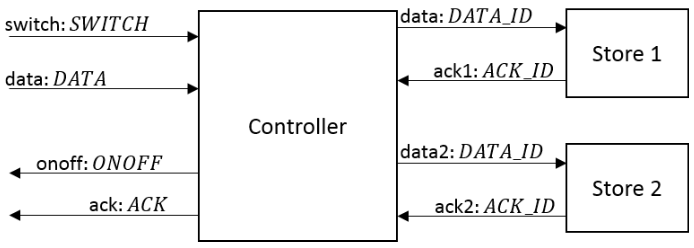


Figure 6.4: Logical architecture of the SAS example system. It consists of a controller and two redundant store components.

StoreX

in	data$X$: $DATA\_ID$
out	ack$X$: ACK_ID
local	$v : \mathbb{N}$ **initial** 0
univ	$i, j, k : \mathbb{N}$

Input		Output	
v	dataX	*v'*	ackX
j	(read, i)	j	(j, i)
j	(set(k), i)	k	(done, i)
j	□	j	□

Controller

in	onoff : $ONOFF$
	ack1 : ACK_ID
	ack2 : ACK_ID
out	onoff : $ONOFF$
	data1 : $DATA\_ID$
	data2 : $DATA\_ID$
	ack : ACK

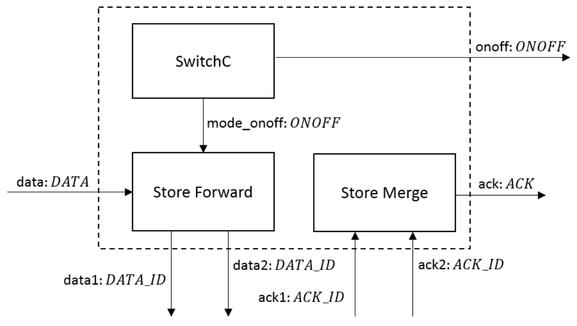


Figure 6.5: Specification of the stores and the controller.

Store Forward

in	mode_onoff : $ONOFF$
	data : $DATA$
out	data1 : $DATA\_ID$
	data2 : $DATA\_ID$
local	$id : \mathbb{N}$ **initial** 0
univ	$j : \mathbb{N}$
	$d : DATA$

Input			Output		
id	mode_onoff	data	*id'*	data1	data2
j	*off*	?	j	□	□
j	*on*	□	j	□	□
j	*on*	d	$j + 1$	(d, j)	(d, j)

Store Merge

in	ack1 : ACK_ID
	ack2 : ACK_ID
out	ack : ACK
local	$id : \mathbb{N}$ **initial** 0
univ	$j, k : \mathbb{N}$
	$a : ACK$

Input			Output	
id	ack1	ack2	*id'*	ack
j	$(a, k)^*$	?	k+1	a
j	?	$(a, k)^*$	k+1	a
j	?	?	j	□

* $k \geq j$

Figure 6.6: Specification of the StoreMerge and StoreForward components.

6.2.3 Technical Architecture

The technical architecture for the SAS example follows mainly its logical architecture. There are three ECUs for each of the main components. Hence, there is one ECU for the controller, and one dedicated ECU for each of the two redundant stores. All ECUs are connected via a network. Figure 6.7 illustrates the technical architecture. For the sake of brevity we omit the detailed specification of the ECUs and the network.

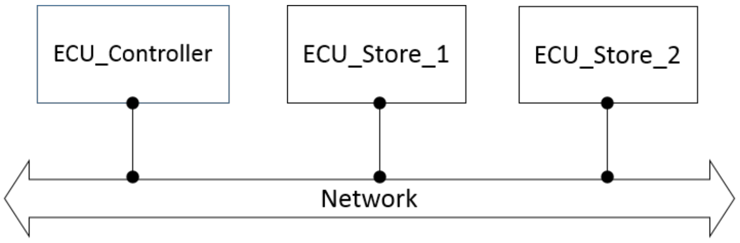


Figure 6.7: Technical architecture of the SAS example system. It consists of three ECUs. One ECU for the controller and one for each redundant store. The ECUs communicate via a common network.

6.3 Basic Building Blocks

In this section, we present a number of generic specifications that we consider useful building blocks or templates for concrete models. Specifically, we provide five models for the basic availability metrics *point availability*, *uptime*, *downtime*, *interval availability* and *steady state availability*. We then extend these simple metrics by a family of more complex specifications that are based on time-slices. Additionally, we present four useful filter specifications to be used in deviation models (*omission*, *delay*, *insertion*, *modification*), three activation functions for deviation models (*non-deterministic*, *probabilistic*, *deterministic*) and a simple comparison component. Table 6.1 gives an overview over the types of basic building blocks we are presenting in the following sections.

6.3.1 Basic Availability Metrics

Point Availability

Point availability $A(t)$ is the probability of a system to be operational at a specific point in time t. This is usually formalized using a random variable X_t where $X_t = 1$ if the system is operational and $X_t = 0$ otherwise. Then, $A(t)$ can be defined as the expected value of X. In our framework, we specify a metric function pa. It takes a failure mode channel fm as input. It is parameterized with the time point t, a type FM capturing the severity levels of the failure mode, and a distinguished severity level $v \in FM$ representing the severity level indicating that the system is unavailable. The result of pa is either 0 (“unavailable") or 1 (“available”).

Availability Metrics	**Filter**	**Activation**	**Comparison**
point availability	*omission*	*non-determinisitc*	*comparator*
uptime	*delay*	*probabilistic*	
downtime	*insertion*	*deterministic*	
interval availability	*modification*		
time-slice availability			
time-slice uptime			
time-slice downtime			
time-slice interval availability			

Table 6.1: Overview over the basic building blocks for availability modeling introduced in this section.

pa $(\mathbf{const}\ t \in \mathbb{N},\ \mathbf{type}\ FM,\ \mathbf{const}\ v \in FM)$ MF

in fm : FM

$$\begin{cases} 0 & \text{if} \quad \mathsf{fm}.t = v \\ 1 & \text{otherwise} \end{cases}$$

Uptime

Expected uptime in a discrete time setting is defined as the expected number of discrete time-units in an interval $[0;t[$, in which the system operates failure-free. Using the specification pa from above, we capture the number of time units the system is operating during $[0;t[$ in the metric function ut, by summing the results of pa for all points in the interval $[0;t[$. The upper bound t is a parameter to the specification of ut.

ut $(\mathbf{const}\ t \in \mathbb{N},\ \mathbf{type}\ FM,\ \mathbf{const}\ v \in FM)$ MF

in fm : FM

$$\sum_{k=0}^{t-1} \mathsf{pa}(k, FM, v)(\mathsf{fm})$$

Downtime

The expected downtime is the complement of uptime. This means, it is defined as the expected number of time-slots in an interval $[0;t[$ where the system exhibits a failure. We employ the specification ut for specifying dt, which counts the “down” time-slots.

$\mathsf{dt}\ (\mathbf{const}\ t \in \mathbb{N},\ \mathbf{type}\ FM,\ \mathbf{const}\ v \in FM)$ MF

$\mathbf{in} \quad \mathsf{fm} : FM$

$t - \mathsf{ut}(t, FM, v)(\mathsf{fm})$

Interval Availability

Interval availability is the expected ratio of uptime in some observation period. Intuitively, this is the percentage of uptime in a given interval $[t_0, t_1[$. Therefore, we can again use ut to define a suitable metric function ia. This specification is additionally parameterized by t_0 and t_1 that define the observation period.

$\mathsf{ia}\ (\mathbf{const}\ t_0 \in \mathbb{N},\ \mathbf{const}\ t_1 \in \mathbb{N},\ t_1 \geq t_0,\ \mathbf{type}\ FM,\ \mathbf{const}\ v \in FM)$ MF

$\mathbf{in} \quad \mathsf{fm} : FM$

$$\frac{\mathsf{ut}(t_1, FM, v)(\mathsf{fm}) - \mathsf{ut}(t_0, FM, v)(\mathsf{fm})}{t_1 - t_0}$$

Steady State Availability

The limiting value of interval availability for infinitely growing observation intervals is called the *steady state availability*. We obtain a suitable metric function ssa through the pointwise limit of ia.

$\mathsf{ssa}\ (\mathbf{type}\ FM,\ \mathbf{const}\ v \in FM)$ MF

$\mathbf{in} \quad \mathsf{fm} : FM$

$$\lim_{t \to \infty} \mathsf{ia}(0, t, FM, v)(\mathsf{fm})$$

Continuity and Measurability

All metric functions, except the one used for steady state availability (ssa) are continuous functions. They are continuous because they are obviously time-bounded. As they are continuous functions, they are also measurable, which is our minimal criterion. The last function, ssa, is not continous. However, as it is the pointwise limit of a sequence of measurable functions, according to Theorem 1 in Section 4.1.1, it is also measurable.

6.3.2 Time-Slice Availability Metrics

In the availability metrics defined so far, we evaluate, for every logical time unit, whether a failure is present and calculate the metrics based on the single time points. However, in many cases, evaluating availability based on single time points is too fine-grained. In practice, it is common to evaluate, whether a failure (or a certain number of failures) occurred in a certain time interval. If yes, the whole interval should be considered failed. Such an approach is employed, for example, in the ITU G.826 guideline (ITU, 2002) describing the availability of certain digital network connections. The guideline defines three basic notions:

Block: A block is a set of consecutive bits.

Errored block (EB): A block in which one or more bits are in error.

Errored second (ES): A one-second period with one or more errored blocks.

Severely errored second (SES): A one-second period that contains ≥30% errored blocks.

In this definition, time is sliced into periods of one second. If a certain number of failures occur in a second, this second is labeled errored or severely errored. The availability definition then builds upon these notions to define available and unavailable time periods. A similar idea is expressed in the availability definition part of the Service Level Agreement of the Amazon S3 file storage webservice (Amazon, 2015), where failures are accumulated over 5 minute intervals and then a failure rate for each interval is calculated. These two examples motivate the adaption of the simple availability metrics definitions. Instead of single logical time units, we change the definitions in a way that they relate to time-slices. A time-slice is essentially a time interval. We prefer the term "slice" here in order to differentiate it from intervals as in "interval availability". We count time-slices starting from 0. This means, the n-th time slice refers to the logical time interval $[n \cdot S, (n+1) \cdot S[$, where S is the slice size.

Time-Slice Availability

Time-slice availability is the analogon to point availability, however, based on time slices. It is parameterized by S, denoting the slice size (i.e. the number of logical time units that a slice spans) and n, the number of the considered slice. It returns 0 if there is a failure in the n-th time slice. For the specification tsa for time-slice availability we use the specification ia of interval availability: If the interval availability in a slice is smaller than one, that means a failure has occurred and we count the whole slice as failed.

$\mathsf{tsa}\ (\mathbf{const}\ S \in \mathbb{N},\ \mathbf{const}\ n \in \mathbb{N},\ \mathbf{type}\ FM,\ \mathbf{const}\ v \in FM)$ MF

$\mathbf{in}\quad \mathsf{fm} : FM$

$$\begin{cases} 0 & \text{if} \quad \mathsf{ia}(n \cdot S, (n+1) \cdot S, FM, v)(\mathsf{fm}) < 1 \\ 1 & \text{otherwise} \end{cases}$$

Based on the notion of time-slice availability we can easily define the notions of time-slice uptime, time-slice downtime and time-slice interval availability.

Time-Slice Uptime

Time-slice uptime denotes the number of non-failed time-slices up to the n-th time slice, where n is a parameter. A second parameter is again S denoting the size of the time-slice. Time-slice uptime is calculated by summing over the time-slice availabilities (which are either 0 or 1) of all time-slices up to (but not including) the n-th slice.

tsu $(\textbf{const } S \in \mathbb{N},\ \textbf{const } n \in \mathbb{N},\ \textbf{type } FM,\ \textbf{const } v \in FM)$ MF

in fm : FM

$$\sum_{k=0}^{n-1} \mathsf{tsa}(S, k, FM, v)(\mathsf{fm})$$

Time-Slice Downtime

The counterpart of time-slice uptime is time-slice downtime. This metric denotes the number of failed time-slices up to a certain slice number. Equally to time-slice uptime, it is parametrized with the slice-length S and the number n of time-slices to consider. We define time-slice downtime by subtracting time-slice uptime from the number of considered slices.

tsd $(\textbf{const } S \in \mathbb{N},\ \textbf{const } n \in \mathbb{N},\ \textbf{type } FM,\ \textbf{const } v \in FM)$ MF

in fm : FM

$$n - \mathsf{tsu}(S, n, FM, v)$$

Time-Slice Interval Availability

Time-slice interval availability extends regular interval availability with time-slices. It is defined as the percentage of failed time-slices with respect to all time slices in a certain interval of slices $[n_0, n_1[$. We use time-slice uptime for the definition of time-slice interval availability.

tsia $(\textbf{const } S \in \mathbb{N},\ \textbf{const } n_0 \in \mathbb{N},\ n_1 \in \mathbb{N},\ n_0 \le n_1,\ \textbf{type } FM,\ \textbf{const } v \in FM)$ MF

in fm : FM

$$\frac{\mathsf{tsu}(S, n_1, FM, v)(\mathsf{fm}) - \mathsf{tsu}(S, n_0, FM, v)(\mathsf{fm})}{n_1 - n_0}$$

6.3.3 Deviation Filters

Recall from Section 4.1.5 that deviation models consist of filters and activation functions. In the following, we present a set of filters that form a basic library to create deviation models. The filters are domain agnostic, that means they do not relate to a specific (type of) functionality and can be employed in any setting. However, more specific or adapted filters may be necessary in specific situations. All the filters that we define here can be used as input and output filters. Each filter defines an activation input channel a, which triggers filter activation. All filters are parameterized by a number of channels n and channel types $T_0, \dots, T_n$. Hence, they can be employed in a great number of contexts.

Omission

When activated, the omission filter removes the current message from all streams. More specifically, it replaces a non-empty message with the empty message □. When not activated, the omission filter forwards the incoming messages unchanged. In the specification, we use a fixed set of activation modes $ACTIVE = \{\mathsf{active}\}$.

$\mathsf{omm}\ (\mathbf{const}\ n \in \mathbb{N},\ \mathbf{type}\ T_0, \dots, T_n)$

in	$\mathsf{a} : ACTIVE$
	$i_1 : T_0, \dots, i_n : T_n$
out	$o_1 : T_0, \dots, o_n : T_n$

$$\forall 1 \le k \le n, t \in \mathbb{N}_0 : (\mathsf{a}.t = \mathsf{active} \wedge o_k.t = \square) \vee (\mathsf{a}.t \ne \mathsf{active} \wedge o_k.t = i_k.t)$$

Insertion

The insertion filter inserts a message into a stream. More specficially, it replaces the empty message □ by a non-empty message but does not modify or delay messages. Other than for the filter omm we do not use a fixed set of activation modes. Instead the filter is parameterized by a set of activation modes A. An additional parameter is a family of decision functions d_i that describe which messages can be inserted for each activation mode. Note that an active insertion filter does not need to have any effect. This is the case, if either a non-empty message arrives at the filter when it is active or the decision function does not provide any insertion alternative for a given activation mode (i.e $d_i(\cdot) = \varnothing$).

$\mathsf{ins}\ (\mathbf{type}\ A,\ \mathbf{const}\ n \in \mathbb{N},\ \mathbf{type}\ T_0, \dots, T_n,\ \mathbf{const}\ d_i : A \to \wp(T_i))$

in	$\mathsf{a} : A$
	$i_1 : T_1, \dots, i_n : T_n$
out	$o_1 : T_1, \dots, o_n : T_n$

$$\begin{aligned}&\forall 1 \le k \le n, t \in \mathbb{N}_0 : [\\ &\quad i_k.t = \square \wedge (o_k.t \in d_k(\mathsf{a}.t) \vee (d_k(\mathsf{a}.t) = \varnothing \wedge o_k.t = \square))\\ &\quad \vee ((i_k.t \ne \square \vee \mathsf{a}.t = \square) \wedge o_k.t = i_k.t)]\end{aligned}$$

Modification

The modification filter replaces a message with a different message. It does not insert any new messages into the stream. Hence, it only takes effect if there already is a message present. Similar to the insertion case, the filter is parameterized by a set of activation modes A as well as a family of decision functions d_i that describe how a message can be substituted for a given activation mode. An active modification filter does not need to have any effect. This is the case, if there is no substitutable message arriving, or if the decision function does not provide an alternative for substitution (i.e. $d_i(\cdot, \cdot) = \varnothing$) for a given input message and activation mode.

$\mathsf{mod}\ (\textit{type}\ A,\ \textbf{const}\ n \in \mathbb{N},\ \textbf{type}\ T_0,\dots,T_n,\ \textbf{const}\ d_i : A \times T_i \to \wp(T_i))$

in $a : A$

$i_1 : T_1, \dots, i_n : T_n$

out $o_1 : T_1, \dots, o_n : T_n$

$$\forall 1 \le k \le n, t \in \mathbb{N}_0 : [\ i_k.t \ne \square \wedge (o_k.t \in d_k(\mathsf{a}.t, i_k.t) \vee (d_k(\mathsf{a}.t, i_k.t) = \varnothing \wedge o_k.t = i_k.t)) \vee ((i_k.t = \square \vee a.t = \square) \wedge o_k.t = i_k.t)]$$

Delay

The delay filter introduces delays into the stream as soon as it is activated. The delay only takes effect, if there is currently a message being received. The filter guarantees that no message is dropped ($i_k \sim o_k$) and it guarantees further that, when the filter is active, no message passes the filter ($a.t = \mathsf{active} \Rightarrow o_k.t = \square$). In order for this specification to be consistent we need to guarantee that there is enough time where the filter is not active to output all delayed messages. Therefore, we demand that the filter must be deactivated for an infinite number of time units. Note that a weaker assumption is possible by considering also the number of messages on the inputs, but we accept the stronger assumption here.

$\mathsf{del}\ (\textbf{const}\ n \in \mathbb{N},\ \textbf{type}\ T_0,\dots,T_n)$

in $\mathsf{a} : ACTIVE$

$i_1 : T_1, \dots, i_n : T_n$

out $o_1 : T_1, \dots, o_n : T_n$

$$|\{\square\} \circledS \mathsf{a}| = \infty \Rightarrow \forall 1 \le k \le n : (i_k \sim o_k \wedge (\mathsf{a}.t = \mathsf{active} \Rightarrow o_k.t = \square))$$

Identity

Finally, the identity filter does not change the input at all. Instead, it just copies the messages it receives. This filter serves as a placeholder in deviation models that possess only an input filter or only an output filter, but not both.

$\mathsf{id}\ (\textbf{const}\ n \in \mathbb{N},\ \textbf{type}\ T_0,\dots,T_n)$

in $i_1 : T_1, \dots, i_n : T_n$

out $o_1 : T_1, \dots, o_n : T_n$

$$\forall 1 \le k \le n\ :\ o_k = i_k$$

6.3.4 Deviation Activation

The second constituent of deviation models are activation functions. We provide three basic specifications for activation functions. All describe a simple activate & deactivate pattern. The first activation function represents non-deterministic activation. The second is a deterministic, non-probabilistic activation that is triggered via a coordination channel. The third provides probabilistic activation and is parameterized with fixed activation and deactivation probabilities.

Non-deterministic Activation

The non-deterministic activation function is parameterized by a type A representing the different possible activation modes. It can switch almost arbitrarily between these modes. However, we demand that deactivation is signaled infinitely often. Thus, we can use this activation function also in conjunction with the delay filter. Another constraint is the equality between the activation channel and the outgoing coordination channel.

ndad $(\textbf{type } A)$

out $\mathsf{a} : A$
$\mathsf{c}_{out} : A$

$\mathsf{a} = \mathsf{c}_{out} \land |\{\square\} \circledS \mathsf{a}| = \infty$

Externally Triggered Activation

The externally triggered activation function is completely triggered from the outside. It therefore includes an incoming coordination channel c_{in}. All it does, is forwarding the activation messages it receives via activation channel a.

tad $(\textbf{type } A)$

in $\mathsf{c}_{in} : A$
out $\mathsf{a} : A$

$\mathsf{a} = \mathsf{c}_{in}$

Probabilistic Activation

The probabilistic activation defines a probability distribution for the activation. This probability distribution is determined by the parameters ap and dp. Parameter $ap(x)$ denotes the probability that an activation with activation mode x takes place from an inactive state, while parameter $dp(x)$ denotes the probability for deactivation from a state with activation mode x. The current activation state is saved in the local variable s. Note that we excluded 0 as probability from dp. This is again to ensure that the filter will be deactivated infinitely often. As the non-deterministic variant, the probabilistic activation includes an outgoing coordination channel c_{out}.

pad $(\textbf{type } A,\ \textbf{const } ap : (A \cup \{\square\}) \to [0,1],\ \textbf{const } dp : A \to]0,1])$

out	$\mathsf{a} : A$
	$\mathsf{c}_{out} : A$
local	$s : A \cup \{\square\}$ **init** $\square$
univ	$v : A \cup \{\square\}$
	$w : A$

Input	Output			
s	a	c_{out}	s'	Prob
$\square$	v	v	v	$ap(v)$
w	w	w	w	$1 - dp(w)$
w	$\square$	$\square$	$\square$	$dp(w)$

6.3.5 Comparator

Later in this chapter, we need a component that detects differences between input streams. The specification comp describes a comparison between pairs of input streams and signals whether the pairs differ by sending the message residual on the channel r. The specification is parameterized by the number and types of input channels. In the specification, we use the type *RESIDUAL* defined as

$$RESIDUAL = \{\mathsf{residual}\} \ .$$

The specification of the comparator comp is as follows

comp $(\textbf{const } n \in \mathbb{N},\ \textbf{type } T_1, \dots, T_n)$

in	$i_1 : T_1, \dots, i_n : T_n$
	$c_1 : T_1, \dots, c_n : T_n$
out	$\mathsf{r} : RESIDUAL$

$$\forall 1 \le k \le n\ \forall t \in \mathbb{N}_0\ :\ i_k.t \ne c_k.t \Leftrightarrow \mathsf{r}.t = residual$$

The presented building blocks are only a small set that, however, proved to be useful for the examples in this thesis as well as for the case study that we will present later. However, this set can be extended by an organization to form a comprehensive library that supports the definition of availability models.

6.4 Systematic Creation of Availability Models

In Chapter 5, we introduced a set of model artifacts, which can be used in combination to model important aspects relevant to availability and to drive the availability analysis. We introduced the formal foundation of these models and discussed description techniques. In this section, we describe how these models can be created systematically. To this end, in Section 6.4.1, we first suggest a modeling process. This process describes how the contents of the different artifacts are created one after another. Afterwards, in sections 6.4.2 to 6.4.4, we propose methods for the creation of each type of model. These methods take the form of step-by-step guides or modeling patterns.

6.4.1 Modeling Process

The artifact model, outlined in Chapter 5 does not prescribe a specific process to create the different artifacts. It only fixes a set of models used to describe the contents of the artifacts. However, due to the relationships between the models, not all processes are equally suitable. For example, as the failure definition model references the failure modes documented in the failure mode list, there is a dependency between these models. It is preferable to create the failure mode list first, in order to profit from the collected information and to avoid unnecessary revisions. However, even though such dependencies exist, there are still various degrees of freedom regarding the sequence in which to create the models and hence different processes are possible. For example, the extended logical architecture is largely independent from the availability specification. Therefore, it is in principle possible to create this artifact before, after or in parallel to the creation of the availability specification. Additionally, it is possible to create the models in an incremental way. An example would be to create all of the necessary models, but only for one function. In a second increment, the models are then extended for a second function, and so forth. Similar, an iterative process with several revisions is possible.

In this section, we outline one specific instance of an availability modeling process based on the availability artifact model. It is a rather simple instance, without increments or iterations. Nevertheless, it proved feasible in our experience. Figure 6.8 gives an overview over the process. The process starts with eliciting the initial, informal availability requirements in the form of textual availability descriptions. The next steps are the creation of the extended logical architecture and the environment specification. During this step, the engineer already gathers an understanding about possible failure modes of the system. This information is helpful for creating the availability specification, which is defined subsequently. Note, that for the availability specification, we first create the failure definition model and the availability metrics model and only afterwards create the aggregation model. The reason for this is that the aggregation model serves as a mediator between the failure definition model and the availability metrics model. Therefore, only after defining these two models we have enough information to specify the aggregation model. In a final step, the formalized availability requirements are specified in the form of the availability constraints model.

6.4.2 Textual Availability Descriptions

We introduced *textual availability descriptions* in Section 5.5.1. They capture availability requirements informally with natural language. They are suited especially for initial development phases. Availability requirements, as any other requirements, are the result of requirements

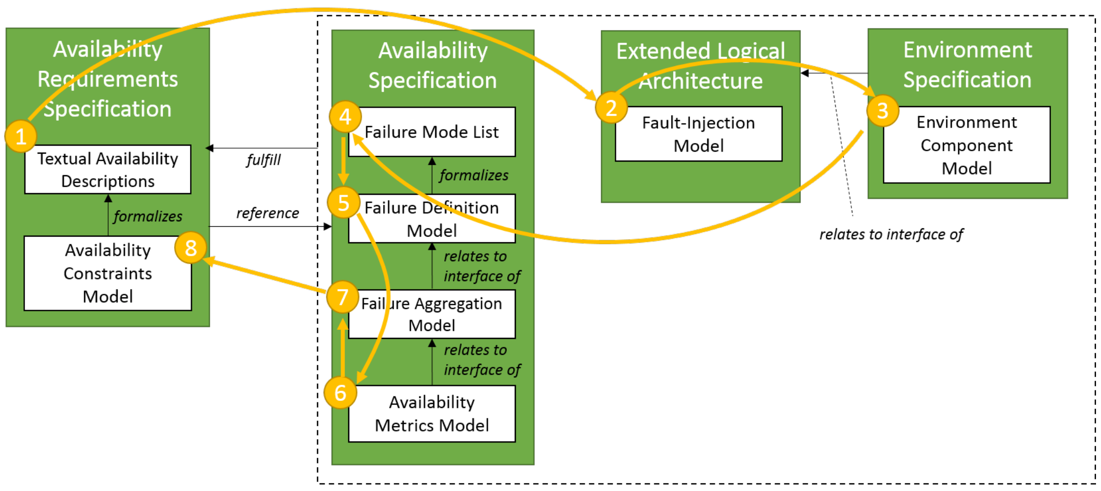

Availability Requirements Specification
Textual Availability Descriptions
formalizes
Availability Constraints Model
fulfill
reference
Availability Specification
Failure Mode List
formalizes
Failure Definition Model
relates to interface of
Failure Aggregation Model
relates to interface of
Availability Metrics Model
Extended Logical Architecture
Fault-Injection Model
Environment Specification
Environment Component Model
relates to interface of
1
2
3
4
5
6
7
8

Figure 6.8: Process for creating the artifacts of the availability artifact model.

elicitation, analysis and documentation. Providing a full-fledged method for requirements engineering is out of scope of this thesis. Detailed guides for requirements engineering can be found in the literature, for instance in the publications by Sommerville and Sawyer (1997) or Van Lamsweerde (2001). Below, we discuss some specifities for availability.

For *requirements elicitation* the relevant stakeholders need to be identified and their demands and interests regarding availability need to be collected. An important stakeholder in the case of availability are the customers and the users of the system. Their needs are often driven by economic considerations, as an unavailable system causes costs. A second group of stakeholders that should be considered are regulation bodies. Further sources for availability requirements are the market situation and competitor products. In some markets a certain level of availability is customary. (IEC, 2006)

The goal of *requirements analysis* is to obtain a complete, consolidated and consistent set of requirements that all stakeholders agree upon (Sommerville and Sawyer, 1997). During the analysis we identify, for instance, conflicting, unnecessary or ambiguous requirements. In case of textual availability descriptions, the analysis is largely done manually by requirements engineers. A helpful instrument for the manual inspection are checklists. A generic requirements analysis checklist is given by Sommerville and Sawyer (1997). In Table 6.2 we extend and specialize Sommerville's checklist for availability requirements.

The textual availability description is produced during *requirements documentation*. It is informal and written in natural language. Although this allows for a large freedom in how to write textual availability descriptions, adhering to *requirement patterns* is considered a good practice. Withall (2007) presents a requirements pattern catalog, including patterns for availability. The basic availability pattern due to Withall, with a slight modification, is

Availability Requirements Pattern

The «**item**» shall normally be available to users «**Availability extent description**» [, except in «**exceptional circumstances**»of a frequency and duration not to exceed «**Tolerated downtime qualifier**»]. "Normally available" shall be taken to mean «**Availability meaning**».

An example for the instantiation of this template, adapted from Withall (2007) is:

Checklist Item	Description
Premature design (S)	Does the requirement include premature design or implementation information, for instance fault tolerance mechanisms or redundancies?
Requirements realism (S)	Is the requirement realistic given the technology which will be used to implement the system?
Requirements scope	Does the availability requirement relate to the appropriate scope? For instance, does the availability requirement relate to the whole system while it should relate only to a system function?
Failure definition	Does the requirement specify what should be considered a failure?
Metric type	If the requirement is quantitative, does it specify a concrete metric type (e.g. uptime)?

Table 6.2: Checklist for availability requirements, items marked with (S) stem from the original list by Sommerville and Sawyer (1997)

Availability of Web site

The **dynamic functions of the company's Web site** shall be normally available to visitors **24 hours per day, every day of the year**, except for **unscheduled downtime** of a frequency and duration not to exceed **1 hour per week** (averaged over each calendar quarter) plus **scheduled downtime** not to exceed **one outage per calendar month**. "Normally available" shall be taken to mean that **the dynamic web pages are loaded and the user functions are operating without showing error messages**.

Application to the Running Example

For this example, we assume that the SAS system is employed in a domain where an availability level of 99.5% is customary. Additionally, we demand a higher availability level of 99.9% for the access function. Concrete metrics corresponding to 99.5% and 99.9% are accumulated downtimes per week of less than 50 minutes and 10 minutes respectively. Therefore, we formulate the following textual availability descriptions for our system functions.

- **R1:** The *switch function* shall be normally available 24 hours per day, every day, except for an accumulated unscheduled downtime of 50 minutes per week. "Normally available" shall be taken to mean that the switch function behaves as specified within a 1 minute period.

- **R2:** The *access function* shall be normally available 24 hours per day, every day, except for an accumulated unscheduled downtime of 10 minutes per week. "Normally available" shall be taken to mean that the access function behaves as specified within a 1 minute period.

6.4.3 Fault Injection Models

We introduced fault injection models in Section 5.3. Their purpose is to extend the logical architecture with the system's behavior in the presence of faults. Only if faults and their effects are incorporated into our models, we can obtain meaningful availability analysis results. Faults may be located in the *software* (e.g. software bugs) or in the *hardware* (e.g. erroneous hardware design, or faults due to wear-out). Faults may be introduced during the design and or during the runtime of the system. We model faults not directly but instead model their *effects*. Note that the effect of a fault is not the same as a failure, as the effect may be local and does not necessarily propagate to the system interface. In the following we present a pattern for modeling hardware as well as software faults. In both cases, we use the deviation modeling technique presented in Section 4.1.5.

Modeling Hardware Faults

To model hardware faults, we consider the logical architecture, the technical architecture and the deployment, which maps components of the logical architecture to entities of the technical architecture. For simplicity, we assume that the behavior of a component in the technical architecture is characterized by the deployed logical components. In the following we provide patterns for modeling faults of ECUs and busses. Faults of other technical equipment can be modeled similarly.

Modeling ECU faults. Recall the *deviation model*, introduced in Section 4.1.5, as means to model behavior deviation. To model the effects of a fault of an ECU, we create deviation models for all components deployed on this ECU. We represent the effect of ECU faults by output filters or input filters or both. For example, we model a complete crash of the ECU by an output filter that suppresses every output. Similarly, we model the fault of an ECU's network interface by an input filter, suppressing, delaying or modifying input messages. When an ECU exhibits a fault, this has an immediate effect on all deployed components. To model this synchrony of the fault effects, we coordinate the deviation models of different components deployed on the same ECU using the coordination channels.

Modeling bus faults. Similar, in order to model a fault of a bus, we model the effects of that fault by creating deviation models for the components that are deployed on ECUs that use that bus. As faults of a bus may either affect the transmission of messages or the reception of messages or both, we again use input filters, output filters or both to model suitable behavior deviations. Again, several components may be affected by a bus fault at the same time and thus we synchronize the behavior deviations by their coordination channels.

Modeling Software Faults

Compared to hardware faults, software faults are exclusively design faults. Any fault in a software has been introduced during the software creation. Therefore, if the software itself is deterministic, then the effects of faults occur also deterministically. Hence, modeling software faults by a probabilistic or non-deterministic behavior modification seems inadequate at first sight. However, it has been argued that, although the introduction of faults is deterministic, the activation of faults in software, in many cases, follows a pattern that can be adequately described by probabilistic or non-deterministic models (Eusgeld et al., 2008). The reason for this is that faults are often only effective in a situation that is characterized through a complex condition. For example, some fault may only be effective for a certain valuation of variables or a certain external state (inputs, files, or other external devices). These variables may not even be part of the system model. Therefore, the occurrence of such an activating condition may be only described non-deterministically or probabilistically. This motivates the use of deviation models to capture software faults as well.

The modeling of software faults itself is straight forward. Again, we model not the software faults themselves, but instead concentrate on the effects of these faults (i.e. deviating behavior). We assume, a software component is represented by a component in the logical architecture. By modeling behavior deviations for such a component, we can, similar to the case of hardware faults, model the effect of software faults.

Application to the Running Example

We consider the SAS system and model hardware faults for the two hardware devices ECU Controller and Network. We choose a fault injection model that only includes message omissions as effect. An overview over the final fault injection model is given in Figure 6.9. Below, we describe its constituents in detail.

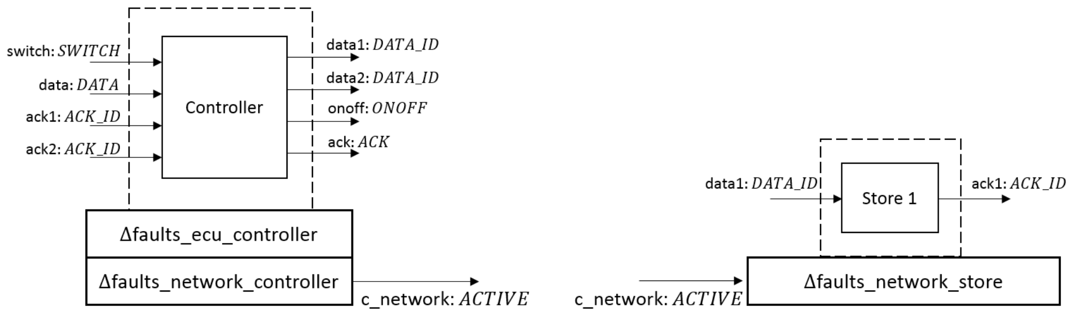


Figure 6.9: Fault injection model for the SAS system showing the deviation models applied to the controller and one of the store components.

Faults of ECU Controller. The only logical component deployed on the ECU Controller is the Controller. Therefore, we only need one deviation model. The deviation model, capturing the fault effects of the controller ECU is given by the tuple

$$\mathsf{faults\_ecu\_controller} = (\mathsf{act}_{ecu}, \mathsf{id}_{ecu}, \mathsf{omm}_{ecu}) \ .$$

The activation function act_{ecu} is derived from the basic building block pad for probabilistic activation. As parameters to the pad specification, we provide the singleton set of activation modes $ACTIVE = \{\mathsf{active}\}$ (see Section 6.3), an activation probability function a_{ecu} and a deactivation probability function d_{ecu}.

$$\mathsf{act}_{ecu} = \mathsf{pad}(ACTIVE, a_{ecu}, d_{ecu}) \ .$$

We choose $a_{ecu} = \{\mathsf{active} \mapsto 10^{-9}, \square \mapsto 1 - 10^{-9}\}$ and $d_{ecu} = \{\mathsf{active} \mapsto 10^{-6}\}$. When we assume that a logical tick corresponds to 10ms, these parameters correspond to a mean-time-to-failure of about 230 days and a mean-time-to-repair of about 5 hours.

The input filter id_{ecu} is derived from the identity building block (i.e. it does not modify the input) with appropriate parameters and by renaming channels.

$$\begin{aligned}\mathsf{id}_{ecu} = \mathsf{id}(4, & DATA\_ID, DATA\_ID, ONOFF, ACK)[\\ & i_1 \mapsto \mathsf{data1}, i_2 \mapsto \mathsf{data2}, i_3 \mapsto \mathsf{onoff}, i_4 \mapsto \mathsf{ack},\\ & o_1 \mapsto \mathsf{data1}', o_2 \mapsto \mathsf{data2}', o_3 \mapsto \mathsf{onoff}', o_4 \mapsto \mathsf{ack}'\\ &]\end{aligned}$$

Finally, the output filter omm_{ecu} is derived from the basic building block omm again by providing parameters and by appropriately renaming the channels.

$$\begin{aligned}\mathsf{omm}_{ecu} = \mathsf{omm}(4, & DATA\_ID, DATA\_ID, ONOFF, ACK)[\\ & i_1 \mapsto \mathsf{data1}', i_2 \mapsto \mathsf{data2}', i_3 \mapsto \mathsf{onoff}', i_4 \mapsto \mathsf{ack}',\\ & o_1 \mapsto \mathsf{data1}, o_2 \mapsto \mathsf{data2}, o_3 \mapsto \mathsf{onoff}, o_4 \mapsto \mathsf{ack}\\ &]\end{aligned}$$

Faults of the Network. More effort is needed for modeling the network faults. In this case, we add a deviation model to every component that accesses the network. In our case, these are the controller and the two stores. If the network breaks down, this affects all communicating components at the same time. To capture this, we have to coordinate the deviation models of the affected components. We realize this coordination with one deviation model for the controller, acting as "master" and deviation models for the stores, acting as "slaves". Slave in this case means, that the activation is triggered externally by the master.

In a first step, we adapt the controller by adding a second deviation model to it. It resembles the first deviation model but instead of the identity input filter it has an omission input filter. With this filter we model the fact that the incoming as well as the outgoing communication is impaired. The deviation model for the controller is given by the tuple:

$$\mathsf{faults\_network\_controller} = (\mathsf{act}_{net\_ctrl}, \mathsf{omm\_in}_{net\_ctrl}, \mathsf{omm\_out}_{net\_ctrl}) \ .$$

The activation component act_{net_ctrl} is also derived from the basic building block pad for probabilistic activation and deactivation. However, we assume, the network is more fragile and more complex to diagnose and repair than the ECU. Therefore we use different probability parameters. We further fix a meaningful name for the coordination channel as this channel will be referenced later by the "slaves".

$$\mathsf{act}_{net\_ctrl} = \mathsf{pad}(ACTIVE, a_{net}, d_{net})[\mathsf{c}_{out} \mapsto \mathsf{c}_{network}] \ ,$$

with $a_{net} = \{\mathsf{active} \mapsto 5 \cdot 10^{-10}, \square \mapsto 1 - 5 \cdot 10^{-10}\}$ and $d_{net} = \{\mathsf{active} \mapsto 5 \cdot 10^{-7}\}$.

The input and output filters are defined in the same way as in the case of faults_ecu_controller and should be sufficiently clear.

In the second step, we add a deviation model faults_network_store to the first of the stores.

$$\mathsf{faults\_network\_store} = (\mathsf{act}_{net\_store}, \mathsf{omm\_in}_{net\_store}, \mathsf{omm\_out}_{net\_store}) \ .$$

This deviation model resembles faults_network_controller. However, as this deviation model should act in the role of the "slave", we employ externally triggered activation instead of probabilistic activation and therefore use the basic building block tad as activation function. We further rename the incoming coordination channel such that it fits the outgoing coordination channel of the "master" (i.e. fault_network_controller).

$$\mathsf{act}_{net\_store} = \mathsf{tad}(ACTIVE)[\mathsf{c}_{in} \mapsto \mathsf{c}_{network}]$$

The input and output filters are again analog to the two deviation models already described. Finally, the deviation model for the second store is equivalent to the model for the first store.

6.4.4 Environment Model

The next artifact we consider is the *environment specification* with its *environment component model*. Having an adequate environment model is important for the precision of the analysis. In several domains, such as automotive control systems, creating environment models is part of the regular development activities, as they are used, e.g. for simulation and verification of the system models. Hence, in these cases, environment models are already given and can be at least partly reused. Possibly, some details need to be added to these models, for example, if they are only specified non-deterministically and not probabilistically. If there are no environment models present, they have to be created. However, a comprehensive method for creating environment models goes beyond the scope of this thesis. In the literature there are several approaches that describe, how to systematically come up with environment models. For example Musa (Musa, 1993, 1996) introduces a step-by-step scheme how to create an environment model (called operational profile in this context) for a system. Typically, requirement documents, such as use cases or scenarios can be consulted for this purpose (Runeson and Regnell, 1998).

In general, it is advisable to create environment models in a modular way. That means, for every external system and every user type (such as train conductor or system administrator), we create a separate component. Every such component mimics the behavior of the corresponding system or user. The environment model is then formed by the composition of the models for the single environment entities. In case of an external system, the corresponding environment model needs to mimic the behavior of the system as well as further systems that are again external to the considered system. In case of a user, the environment model typically captures the workflows that this user performs.

As creating deterministic (non-probabilistic) environment models is often either to expensive (in the case of external systems) or not possible (in the case of user models or models of physical systems), we will, in practice, often use probabilistic models. Shukla (Shukla et al., 2004) suggests to approach the problem of creating probabilistic environment models by first modeling the behavior qualitatively and quantify the behavior afterwards by attaching probabilities. The probability values may stem either from execution traces of the system itself or similar systems (e.g. a legacy system) or they need to be estimated.

6.4.5 Failure Mode List

The *failure mode list* is part of the artifact *availability specification*. We introduced this model in Section 5.2.1. Its purpose is to systematically document and structure the failure modes and the corresponding severity levels that should be considered. Creating such a failure mode list can be challenging, as there usually is a great number of potential failure modes that one can immediately think of. However, the set of failure modes that emerges from such a brainstorming is often unstructured and incomplete. To ease this problem, we suggest a three step approach in which failure modes are collected in a structured way according to the system functions and their interfaces. Using a set of guide words, we first create a list of failure mode candidates, which are reduced in a second step. The third step consists of complementing the list with severity levels.

Step 1: Identification of Failure Mode Candidates with Guide Words

In the literature on the assessment of safety-critical systems, there are a number of suggestions regarding the identification of failure modes. A common approach is to use guide words to list possible failure modes. Such an approach is, for instance, used as part of the safety assessment method HAZOP (Kletz, 1999). A guide word represents a type of deviation of an observed behavior with respect to the expected behavior. In the original HAZOP method, which was developed in the domain of chemical engineering, the list of possible guide words related mainly to material flows and included words such as MORE, LESS, EARLY, LATE. With respect to software-intensive systems, sets of guide words have been proposed, for instance by Bondavalli and Simoncini (1990), as well as by McDermid and Pumfrey (McDermid and Pumfrey, 1994; Pumfrey, 1999) as part of the SHARD method. Both identify roughly the dimensions *value*, *timing* and *omission/insertion* of signals and suggest the set of guide words OMISSION, COMMISSION, EARLY, LATE, SUBTLE, COARSE, where SUBTLE and COARSE relate to the range of the value deviation. A similar classification of failure mode types can be found in works on failure modeling. For instance, Powell (1995) distinguishes on the highest level between failures in the value domain and failures in the time domain.

We adopt the general approach of using guide words to identify relevant failure modes. However, other than for example McDermid and Pumfrey we avoid integrating statements about the failure severity into the guide words (such as COARSE). We use the following guide words to identify failure mode candidates:

OMISSION, INSERTION, EARLY-TIMING, LATE-TIMING, MODIFICATION.

We apply the guide words on the output channels of the system functions, one function at a time. For each combination of output channel and guide word we assess, whether the combination indicates a possible failure mode. Note that the guide words are only an aid to *identify* possible failure modes, but do not fully describe the failure modes. For example, the guide word LATE TIMING does not specify whether the delay relates to calendar time, end-to-end latency, time synchrony with signals of a different channel etc. There may be even more than one failure mode related to the combination of a guide word to an output channel. Therefore, every failure mode identified by the help of the guide words needs to be described to document its intention. As a result of this step, we obtain a partially filled failure mode list with failure mode candidates and descriptions, structured according to functions and output channels.

Step 2: Selection of Relevant Failure Modes

In the previous step we elicited a list of failure mode candidates. However, not all of these failure modes are relevant, for instance because it is known that they cannot occur, or because they do not relate to availability. As an example, consider the case of a control system where the omission of a signal is considered relevant (as some control action is not performed), however the repeated sending of a signal is not relevant (as the control action is idempotent). Therefore, in this step, we inspect the list of failure modes candidates to identify the failure modes that are relevant from an availability perspective. The decision which failure modes are relevant and which are not is system or, at least, domain specific and considers the system's stakeholders and its environment (e.g. the surrounding system). As a result of this step, the initial failure mode list is reduced to only relevant failure modes. Although we described the identification and the relevance assessment as two sequential steps they need not necesarily be performed sequentially, but the two steps can be performed interleaved, which avoids describing an irrelevant failure mode in detail.

Step 3: Specification of Severity Levels

The last step to complete the failure mode list is the specification of the severity levels for the failure modes. Similar to the assessment of the relevance of failure modes, the severity levels are system specific or domain specific and relate to the system's stakeholders and environment. In the simplest case, a failure mode only has one severity level. In other cases a more fine-grained distinction is necessary. Often, a two-fold distinction, such as *non-critical* and *critical*, is sufficient. Consider, for example, a failure mode describing the delay of a signal for a control system. A common distinction is between a short delay (not impairing the control quality) and a critically long delay (impairing the control quality). By extending the partial failure mode list for each failure mode with a list of severity levels, each with an expressive description, we finally obtain the complete failure mode list.

Application to the Running Example

We apply the approach explained above to our example of the SAS system. To this end, we analyze the output and mode channels of the SAS system with respect to our set of guide words. In Figure 6.10, we summarize which failure modes we consider for each channel of the functions Switch and Access.

For the output channel onoff and the output channel ack, we consider failure modes for all guide words except Early-Timing. According to the specifications, the signals sent over these channels are immediate reactions to input signals. To arrive early, the output signals would have to be sent before the stimulus, which would violate our assumption of causality. For the mode channel mode_onoff we do not consider the failure modes for the guide words Insertion and both timing guide words. The reason is, that this channel transmits a state. According to the specification a signal should be transmitted in every time slot. Therefore, insertions as well as timing failure are impossible to distinguish from modifications. Hence, we only consider failure modes relating to Omission and Modification in this case. Figure 6.10 provides informal descriptions of the failure modes that we consider.

The last step consists of the specification of the severity levels. For most failure modes, we only consider one severity level. In three cases, however, we consider more than one level. In case of a delay, we distinguish between a delay with less or more than 0.5 seconds. In case of a modification of the mode_onoff channel, we distinguish between off sent instead of on, or vice versa. In case of a modification of the *ack* channel, we distinguish between the three cases: the value is only off by one, the value is off by more than one, or a number has been replaced with *done* or vice versa. The informal descriptions of the severity levels for all failure modes are given in Figure 6.11. Together, the Figure 6.10 and the Figure 6.11 form the failure mode list for our example.

Function	Channel	Failure Mode	Description
Switch	onoff	*omission*	An on or off signal is completely omitted, although a switch signal has been received.
		insertion	An on or off signal is sent without reception of a preceding switch signal.
		delay	The correct on or off signal is sent, however, not immediately but after a delay.
		modification	Instead of on the signal off is sent or vice versa.
	mode_onoff	*omission*	The current on/off mode is not sent.
		modification	A wrong on/off mode is sent.
Access	ack	*omission*	Upon reception of a read signal, no number is sent or upon reception of a set(·) signal, no done signal is sent.
		insertion	A number or the done signal is sent although no preceding read or set(·) has been received.
		delay	The response on read or set(·) is not sent immediately but with a delay.
		modification	Upon reception of read, a wrong number is sent.

Figure 6.10: Informal description of the failure modes for the SAS example system.

6.4.6 Failure Definition Model

As described in Section 5.2.2, the *failure definition model* extends and formalizes the failure mode list. It represents the informal failure modes and severity levels by failure mode channels and formal channel types. The failure definition model also extends the failure mode list, by describing which observed behavior corresponds to which kind of failure in terms of failure modes and severity levels.

Creating a failure definition is challenging for two reasons. First, creating a failure definition model for the whole system at once is hard. To tame the complexity, a failure definition model needs to be decomposed adequately. Second, a failure definition model needs to be consistent with the functional requirements. For a behavior that conforms to the functional requirements, a failure definition should not detect a failure.

In this section, we suggest a three step approach for creating a failure definition model. The core idea is to decompose the failure definition model according to system functions. In our experience, system functions provide an adequate granularity for describing failure definitions. Therefore, the first step is the decomposition of the failure definition model into function-specific failure definitions. For the second step, the specification of the individual failure definitions, we propose a modelling pattern which reuses the original function specification. In the third step, we integrate the single failure definitions and handle function interaction.

Step 1: Decomposition of the Failure Definition Model

As providing a single failure definition for the whole system is usually too complex, we decompose the failure definition model in a number of smaller failure definitions. We suggest

Channel	**Failure Mode**	**Severity Level**	**Description**
onoff	*omission*	*omission*	Message omitted
	insertion	*insertion*	Message wrongly inserted
	delay	*delay*	Delay below 0.5 seconds
		critical delay	Delay equal to or above 0.5 seconds
	modification	*modification*	Wrong status transmitted
mode_onoff	*omission*	*omission*	Message omitted
	modification	*on→off*	The signal off is sent instead of the signal on
		off→on	The signal on is sent instead of the signal off
ack	*omission*	*omission*	Message omitted
	insertion	*insertion*	Message wrongly inserted
	delay	*delay*	Delay below 0.5 seconds
		critical delay	Delay equal to or above 0.5 seconds
	modification	*small*	A transmitted number is modified by ± 1
		large	A transmitted number is modified by more than 1
		type	The done signal is sent instead of a number or vice versa

Figure 6.11: Informal description of the failure mode severity levels for the SAS example system.

to structure the failure definitions according the system functions. That means, we create one failure definition per system function. The interface of such a failure definitions consists, on the one hand, of the input, output and mode channels of the system function. These serve as inputs to the failure definition. The outputs of the failure definition are the failure mode channels, which we derive from the failure mode list. We already structured the failure mode list according to system functions, therefore the according failure modes can be easily obtained. The remaining step is to specify formal channel identifiers for the failure mode channels as well as to specify formal types for the severity levels and assign them to the failure mode channels.

Step 2: Specification of Failure Definitions

In this step we actually specify the failure definitions' behavior. It should be consistent with the requirements of the system, as well as with the specification of the system functions. Furthermore, a failure definition should account for the fact that different failure modes could be present at the same time. In order to cater for these needs, we suggest a modeling pattern. The core idea of the pattern is to include the function specification into the failure definition and to identify failures based on the deviation of the observed behavior from the specified behavior.

Figure 6.12 gives a schematic overview over the modeling pattern. A failure definition created according to the pattern consists of three parts:

Function Deviations This part of the pattern includes the specification of the original system function and a deviation model. The deviation model modifies the specified behavior and signals what has been modified by its coordination channels. The goal of this part of the model is to produce any possible deviation that fits to some of the failure modes. Therefore, when designing the deviation model, care must be taken that the deviations match the failure modes that should be considered. For instance, if one failure mode relates to the omission of a signal, a deviation model should be included that performs signal omission. The deviation model usually uses a non-deterministic activation.

Comparator The original function, together with the deviation model, produce outputs that deviate from the specification. The comparator compares the resulting outputs with the actually observed outputs. When the observed behavior does not match the behavior produced from the specification with the deviations at a given instance of time, this indicates that either the wrong types of deviations have been chosen non-deterministically or the modelled deviations are not enough. In any case, we interpret this situation as if a failure of a residual, not yet modelled, failure mode has appeared. Therefore, the comparator signals this situation with a dedicated failure mode channel $\mathsf{fm}_{residual}$.

Failure Mode Mapping The task of this part of the modeling pattern is to use the coordination channels of the deviation model in order to populate the failure mode channels. This part of the model is optional. If the coordination channels can be already mapped one-to-one to failure mode channels, it may be omitted. Sometimes, however, it takes the information of several coordination channels to decide if a failure of a certain failure mode is present.

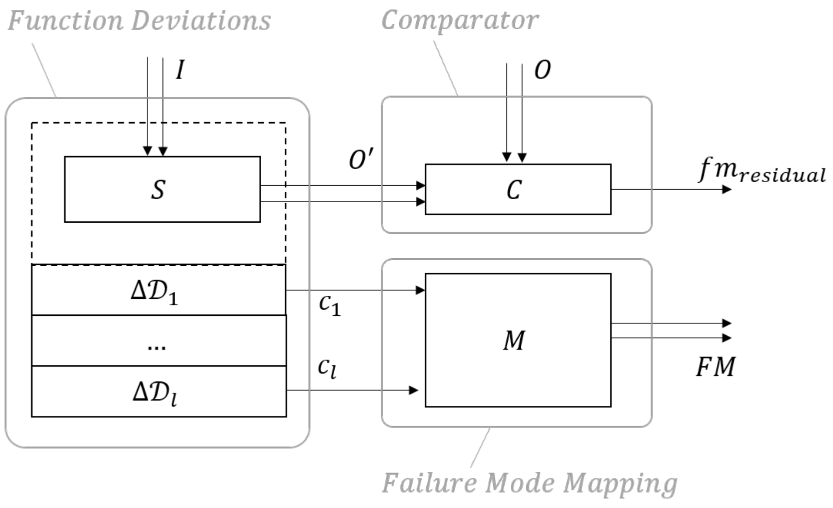


Figure 6.12: Schematic overview over the modeling pattern for a failure definition.

Step 3: Integration of Failure Definitions

At this stage, we obtained a function specific failure definition for every system function. The next task is to integrate these. This integration step is necessary because in multifunctional systems, system functions are often not isolated but instead interact with each other. These interactions are modeled by communication over a mode channel from one system function to the other. These mode channels are also present in the interfaces of the function specific failure definitions. However, most of the time, mode channels are internal and not a part of the system interface.

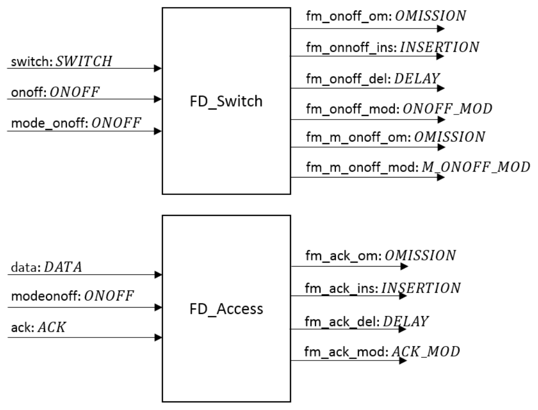
We resolve this situation by introducing further components into the failure definition model that non-deterministically choose values for internal mode channels. Hence, we include both options (failure due to A and failure due to B) as possible interpretations in our model.

Application to the Running Example

We apply the above steps to our running example. In the first step we decompose the failure definition model into two failure definitions FD_Switch and FD_Access, one for each of the system functions. The syntactic interface consists of the input and output channels of the system functions, and the failure mode channels according to the failure mode list (see Figures 6.10 and 6.11). The result of the decomposition is shown in Figure 6.13.

In the next step we specify the behavior of the failure definitions. We illustrate this step by the failure definition for the Switch function, FD_Switch. We use the modeling pattern for failure definitions we introduced above. An overview over the specification of FD_Switch is given in Figure 6.14.

The first part of the modeling pattern is the function deviation part. In case of FD_Switch it contains the original specification of the system function (Switch), To disambiguate the channel names, we rename the output channels of Switch by adding the suffix "_spec". Next, we apply several deviation models to the specification. The names of the deviation models already suggest that every deviation model relates to a failure mode (e.g. the deviation model Ommit_mode_onoff relates to the failure mode fm_m_onoff_om). All deviation models can be



Failure Mode Type	**Severity Levels**
$OMISSION$	{omission}
$INSERTION$	{insertion}
$DELAY$	{delay, critical}
$ONOFF\_MOD$	{modification}
M_ONOFF_MOD	{on_off, off_on}
ACK_MOD	{small, large, type}

Figure 6.13: Decomposition of the failure definition model of the SAS in two separate failure definitions, one for each of the system functions Switch and Access. The interfaces mirror the syntactical interfaces of the functions and the failure modes from the failure mode list. The channel types in the right table represent the severity levels from the failure mode list.

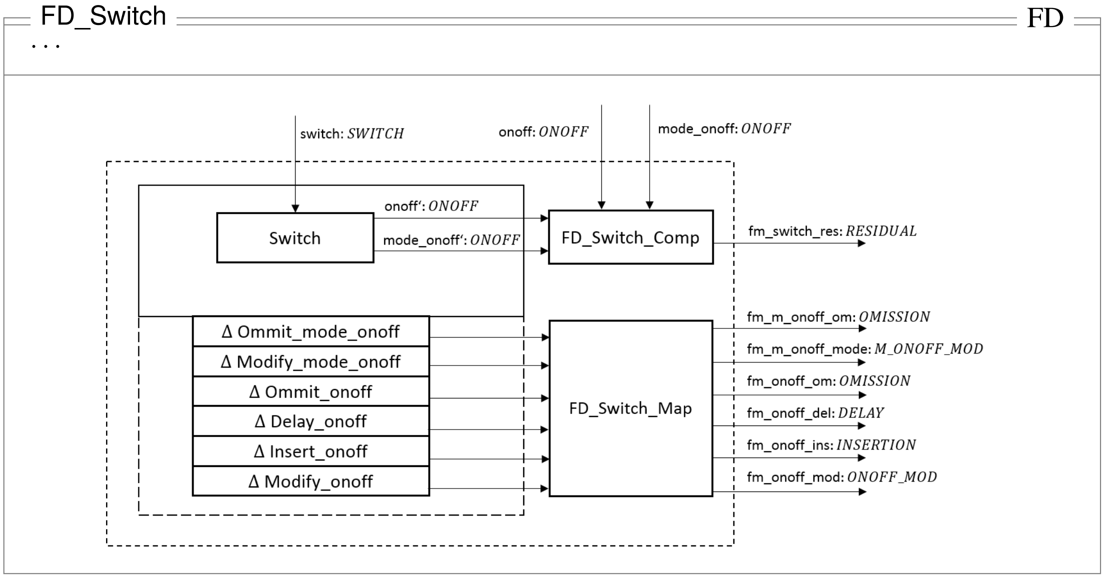


Figure 6.14: Specification of the failure definition for the *Switch* function.

built using the basic building blocks introduced in Section 6.3. As an example, we present details for the deviation model Modify_mode_onoff in Figure 6.15. It uses the non-deterministic activation function ndad as well as the modification filter mod to model modifications of the mode_onoff channel.

Modify_mode_onoff ═══ deviation

act

out co_modify_mode_onoff : M_ONOFF_MOD
a : M_ONOFF_MOD

(co_modify_mode_onoff, a) = ndad(M_ONOFF_MOD)($\mathbf{0}$)

of

in onoff_spec′ : $ONOFF$
mode_onoff_spec′ : $ONOFF$
a : M_ONOFF_MOD
out onoff_spec : $ONOFF$
mode_onoff_spec : $ONOFF$

$$\text{let } d(x,y) = \begin{cases} \{\text{on}\} & \text{if} \quad x = \text{off\_on} \wedge y = \text{off} \\ \{\text{off}\} & \text{if} \quad x = \text{on\_off} \wedge y = \text{on} \\ \varnothing & \text{otherwise} \end{cases}$$

in
mode_onoff_spec =
mod(M_ONOFF_MOD, 0, $ONOFF$, $ONOFF$, d)(mode_onoff_spec′)
$\wedge$ onoff_spec = onoff_spec′

Figure 6.15: Specification of the deviation model Modify_mode_On_Off. Its activation component uses the building block *ndad* (non-deterministic activation and deactivation). The output filter uses the parameterized *mod* building block to describe the modification.

The second part of the pattern is the failure mode mapping. In case of FD_Switch, there is a one-to-one correspondence between the coordination channels of the deviation models and the failure mode channels. Therefore, the mapping performed in FD_Switch_Map is extremely simple and we omit the details.

Finally, the third part of the pattern comprises the comparison between the modified outputs and the observed outputs. This is realized in the component FD_Switch_Comp. This component is merely an instantiation of the basic building block comp, as its specification in Figure 6.16 shows.

The last step is the integration of the two individual failure definitions. In the SAS system, we have a function interaction between the two system functions via the mode channel mode_onoff. This channel is internal to the system and not part of the system interface. To reflect this in our

FD_Switch_Comp

in onoff_spec : $ONOFF$
onoff : $ONOFF$
mode_onoff_spec : $ONOFF$
mode_onoff : $ONOFF$
out fm_switch_res : $RESIDUAL$

fm_switch_res = comp$(2, ONOFF, ONOFF)$(onoff_spec, mode_onoff_spec, onoff, mode_onoff)

Figure 6.16: Specification of the comparator for the failure definition FD_Switch_Comp.

failure definition model we add a further component nd_mode_onoff. Its only output channel is mode_onoff and its specification is *true*, modeling the non-deterministic choice for the mode channel (see Figure 6.17). The final failure definition model for the SAS system, including the failure definitions obtained from the functions and the non-deterministic choice for the mode channel is depicted in Figure 6.18.

nd_mode_onoff

out mode_onoff : $ONOFF$

$true$

Figure 6.17: Specification of non-deterministic choice of values for the internal mode channel.

6.4.7 Availability Metric Model

The next model that we consider is the availability metric model. We introduced this model type in Section 5.2.4. With this model we describe calculation rules for availability metrics. In general, all modeling techniques presented in Chapter 5 can be employed for this purpose but in many cases, a standard metric can be used (such as the basic building blocks in Section 6.3).

To find an appropriate metric type, we suggest to first consult the informal availability requirements, if any availability metrics are mentioned there. If this is not the case, other means have to be taken to select the metric type, for example performing interviews with the relevant stakeholders or identify metric types that are customary in the domain.

Application to the Running Example

We are going to specify availability metrics for the two functions of the SAS system. In the informal requirements of the SAS system we stated a maximum weekly downtime of 50 minutes for the Switch function and a maximum weekly downtime of 10 minutes for the Access function. The informal requirements further stated that 1-minute-periods should be used for the assessment. The appropriate basic building block to capture the informal requirements is the

FD_SAS

. . .

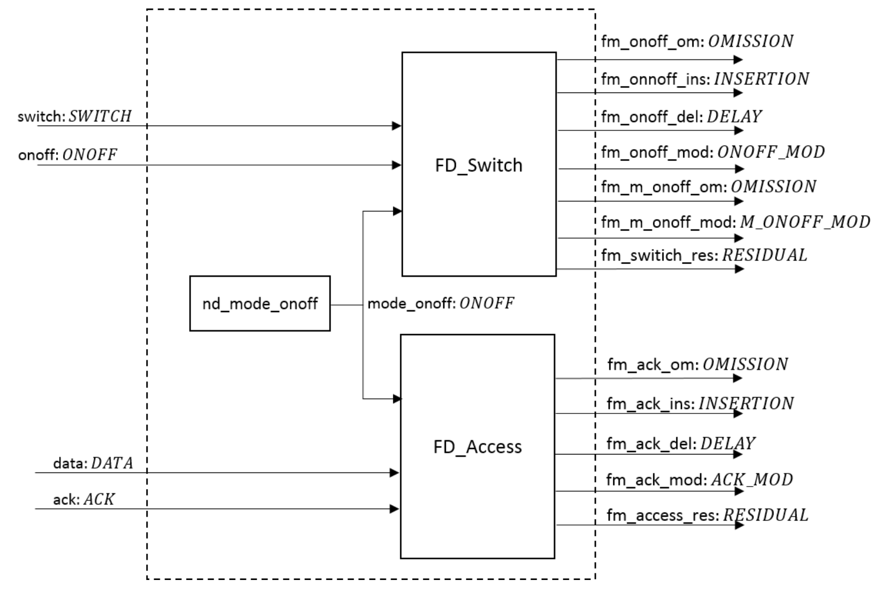


Figure 6.18: The complete specification of the failure definition for the SAS system.

time-slice downtime with 1-minute slice length. For this we introduced the specification tsd in Section 6.3. To apply the specification, we define the following singleton set of severity levels: $FAIL = \{\mathsf{fail}\}$. We assume, a logical time unit corresponds to 200ms and fix the following constants:

- $sl = \frac{1000ms}{200ms} \cdot 60$ is the slice length of one minute in logical time units (each assumed to correspond to 200ms).
- $week = \frac{1000}{200} \cdot 3600 \cdot 24 \cdot 7$ is the number of slices (i.e. seconds) in one week.

We now define the availability metric Downtime_Access for the Access function as follows. The availability metric Downtime_Switch for the Switch function is defined accordingly.

Downtime_Access MF

in fail_access : $FAIL$

tsd$(sl, week, FAIL, \mathsf{fail})(\mathsf{fail_access})$

6.4.8 Failure Aggregation Model

The last model type in the artifact availability specification is the aggregation model. We introduced it in Section 5.2.3. Its purpose is to adapt the failure modes documented in the failure mode list and the failure definition model, to the chosen availability metric model. These models are usually rather simple and do not require an involved method. Therefore, we immediately illustrate this model type with our running example.

Application to the Running Example

In our case, the availability metrics require only one failure mode (fail_access and fail_switch, respectively) and a reduced set of severity levels $FAIL = \{\mathsf{fail}\}$. Hence, we need to merge all failure modes, we defined in the failure definition model, into one failure mode with only one level of severity. To achieve this, we devise two aggregators. For the Access function the according specification is depicted in Figure 6.19. Although the specification description is lengthy, the idea behind it is simple: As soon as one failure mode indicates a failure, then the combined failure mode fail_access signals failure. There are two exceptions to this rule. We choose to ignore non-critical delays (fm_ack_del = delay) as well as small modifications of the stored number (fm_acc_mod = small). As the aggregation of failure modes for the Switch case is similar, we omit the details here.

Agg_Access

in fm_ack_om : $OMISSION$
fm_ack_del : $DELAY$
fm_ack_ins : $INSERTION$
fm_ack_mod : ACK_MOD
fm_access_res : $RESIDUAL$
out fail_access : $FAIL$

fail_access = fail ⇔ fm_ack_om = omission ∨ fm_ack_del = critical ∨ fm_ack_ins = insertion
∨ fm_ack_mod ∈ {large, type} ∨ fm_access_res = residual

Figure 6.19: Aggregation of failure modes and severity levels for the Access function.

6.4.9 Availability Constraints Model

Finally, we create the availability constraint model. With this model, we express the informal availability requirements in a precise and formal way be referring to the availability metrics model. As outlined in Section 5.5.2, availability constraints are pairs of an availability metric and a value range. To create the availability constraints model, we need to translate the informal requirements to these pairs. As the availability metrics models are developed with the original requirements in mind, this translation is often straightforward.

Application to the Running Example

In our example, we initially stated two informal requirements (see Section 6.4.2). Requirement R1 limits the downtime of the Switch function to 50 minutes and R2 limits the downtime of the Access function to 10 minutes. We precisely defined the calculation rule for the downtime with respect to the SAS system in Section 6.4.7, where we specified the two availability metrics

Downtime_Access and Downtime_Switch. We now use these two availability metrics for the following two availability constraints C_1 and C_2, formalizing R1 and R2:

$$C_1 = (\mathsf{Downtime\_Access}, [0, 50])$$
$$C_2 = (\mathsf{Downtime\_Switch}, [0, 10])$$

6.5 Summary

In this chapter, we complement the artifact model presented in Chapter 5 with a modeling method. The method comprises

- a concrete *process* for instantiating the artifact model,
- *basic building blocks* to be used for assembling failure definition models and availability metric models,
- *step-by-step guides* for developing the failure mode list and the failure definition model, and
- *modeling patterns* for failure definitions and fault-injection models.

We demonstrate the method using the running example of the Storage and Access System (SAS), introduced by Broy (Broy, 2010b).

6.5.1 Basic Building Blocks

Basic building blocks are small, generic specifications that can employed in the specification of concrete availability models. In Section 6.3, we present a set of such basic building blocks. The basic building blocks include specifications for availability metrics, deviation model filters and deviation model activators. The building blocks are parametrized, such that the number of channels and channel types can be easily adapted in concrete specifications. The basic building blocks for availability metrics include simple availability metrics evaluating the availability for every logical time unit, but also more complex ones, evaluating the availability for time-slices of configurable length. The second type of basic building blocks are filters for deviation models. They cover signal omission, insertion, modification and delay. Additionally we describe an identity filter that performs no modification and can be used as placeholder in deviation models. The third type of building blocks are activation functions for deviation models. We introduce specifications for non-deterministic activation, for deterministic activation and for externally triggered activation.

6.5.2 Modeling Process

The artifact model described in Chapter 5 does not prescribe a fixed sequence for creating the models. Nevertheless, because of relationships between models not each such sequence is suitable. In Section 6.4.1, we suggest a process for sequentially creating the different models that proved feasible in our experience. It is a simple process without increments and iterations. It starts with the informal availability requirements and continues with the extension of the logical architecture with faults and the environment model. Afterwards we create the availability specification and finally formalize the informal availability requirements with availability constraints. Throughout the chapter we follow this process and apply it to the running example.

6.5.3 Step-by-Step-Guides

The modeling process provides a path through the artifact model during the development. However, it does not provide guidance for creating the individual models. We give such guidance with step-by-step-guides for the failure mode list and the failure definition model. They allow for the systematic creation of these models. For the failure mode list we furthermore suggest a guide word based technique for the elicitation of failure modes. Such a technique has been already employed for eliciting failure modes in the safety domain. For the failure definition model we suggest a decomposition scheme that follows the functional decomposition in the functional architecture and an integration scheme that is able to handle function interaction.

6.5.4 Modeling Patterns

Patterns provide a basic structure for models and capture best practices. In this chapter we introduce two such patterns. In Section 6.4.6 we describe a modeling pattern for failure definitions. The pattern promotes the re-use of the original function specification and hence fosters consistency beteen the failure definition and the functional requirements. In Section 6.4.3 we provide a second pattern for the creation of fault-injection models. The pattern captures the effect of different types of hardware faults (ECUs, busses) in terms of changed behavior of the deployed components.

Chapter 7

Case Study: Train Door Control

In this chapter, we present a case study that we performed in cooperation with an industry partner. The goal of this case study is to validate the artifact model and the modeling method presented in chapters 5 and 6. With the case study we investigate whether the modeling approach is applicable in an industrial setting, that means if the industrial availability requirements can be captured and if we can perform an automated availability analysis. In this chapter we furthermore report on initial tool support for our approach and measure the performance of automated analyses. Parts of this chapter have been previously published in (Böhm et al., 2014). Note, that some or all products and product names referred to within this chapter, such as Trainguard MT, are protected brands of Siemens or associated companies.

7.1 Context of the Study

The case study has been performed as part of a bilateral research project between Technische Universität München and Siemens AG. The aim of this research project was to demonstrate and evaluate a seamless model-based development approach based on an industrial case example. The branch of Siemens involved in this project is concerned with the development of automatic train control systems. These types of systems are deployed for regional and metropolitan railway systems. They are responsible for the safe and efficient controlling of the railway traffic. They potentially control all automated components of a railway system including trains, tracks, signals, and platform components such as platform doors. In this project, we considered a specific Siemens product called Trainguard MT, a train control system for metro lines, which supports fully automated train operation. As a train control system is a very large system, we considered only a part of the system that is concerned with controlling the train doors as well as the platform doors.

An important selling point of the Trainguard MT is its availability and thus the ability to deliver continuous automated rail operation. The availability analysis in this context comes with an additional challenge. The train control systems are no mass products but instead are configured and customized for every customer. Hence, the availability analysis is currently done for the generic (non customer-specific) system and additionally for every customer-specific system. According to the Siemens engineers, as of today, only few results from the analysis of the generic system can be reused for analysis of a customer-specific system.

7.2 Study Goal and Research Questions

In this section we formally state the study goal from which we derive three research questions to guide the study.

7.2.1 Study Goal

The goal of this case study is to evaluate whether our artifact model and modeling method is applicable for modeling and verifying availability requirements in a realistic, industrial context.

We do not evaluate, whether our approach yields results that match the availability observations in the reality. The reason for this is that we lacked access to a running instance of the system. Furthermore, as we modeled the system according to the specification and after it was already built, we do not know if the implementation matches our models. However, in order to plausibilize our numerical results, we compare them to the results of a commercial analysis tool.

7.2.2 Research Questions

We structure this study using the following three research questions.

RQ1: Modeling Adequacy – Can the industrial availability requirements be modeled with our approach?

By answering this question, we aim to validate whether our artifact model and modeling method fit the purpose to capture industrial availability requirements. Specifically, we want to know if the failure definition model and the availability metric models can be usefully employed. Furthermore, we want to evaluate the guide word method to elicit failure modes, the failure definition template and our various basic building blocks.

RQ2: Analysis Feasibility – Can the created models be automatically analyzed?

With this question, we aim to investigate whether an automated availability analysis according to Section 5.6 can be performed based on our models. Specifically, we are interested in determining if the analysis does indeed provide plausible results and to what degree such an analysis can be performed in terms of analysis time and computing resource consumption.

RQ3: Flexibility – How fragile are our models in the presence of changes in the system?

A specific problem in the context of Siemens is to reuse analysis results obtained from a generic version of the Trainguard MT system for the analysis of a customer-specific system. With this research question, we want to determine if our approach can be beneficially employed in such a setting. More specifically, we aim to investigate the impact of changes in the functionality or architecture of the system to the availability models.

7.3 Data Collection Procedure

In this section we describe which data we use in order to answer the above research questions and how we obtain this data. We perform two high-level steps in this case study. The first step is modeling the TGMT OBCU system including its availability requirements. The second step is performing an availability analysis.

7.3.1 Data Collection for RQ1

To assess whether our artifact model and modeling method are adequate for the case example, we qualitatively evaluate the resulting models and the modeling work we did. We evaluate the models with respect to their completeness (Could the availability requirements be captured?) and the method with respect to its effectivity (Could we create the models using the process, basic building blocks, guides and patterns outlined in the method?). We further presented intermediate models to the Siemens engineers and asked for their feedback.

7.3.2 Data Collection for RQ2

To answer RQ2, we create a prototypical tool chain for automated availability analysis and use it to analyze the models that we created. We record the run time of the analyses and the memory usage. To evaluate whether the results are plausible we compare them with results obtained from an RBD-based analysis using a commercial tool.

7.3.3 Data Collection for RQ3

To evaluate the robustness of our availability models in the case of architecture changes we perform changes on the functional and logical architecture and evaluate which changes of the analysis models are necessary to reflect the architecture changes.

7.4 Study Setup

In this section, we describe the setup of our study. More precisely, we first outline the tool chain that we used for modelling in Section 7.4.1. Afterwards, we introduce the case example and the initial system model that we created for the case example in Section 7.4.2.

7.4.1 Modeling and Analysis Tool Support

The AutoFocus 3 Modeling Tool

We modeled the case example using the tool AutoFocus 3 (AF3)[1]. AF3 is an open-source modeling tool that supports seamless modeling of time-discrete reactive systems, starting with model-based requirements analysis, and including functional, logical and technical architectures.

[1]http://af3.fortiss.org

To create models, several description techniques can be used in AF3. Functional and logical architectures with several levels of hierarchy can be modeled in AF3 using nested data-flow networks. The main structural modeling concept in AF3 is a *component*. A component is equipped with typed input and output *ports*, which can be connected via *channels*. A component can again contain a data-flow network or a behavior description. To model the input/output behavior of a component, AF3 supports a range of behavior description techniques including I/O automata, I/O tables, or textual specifications (called *code specifications*) using a simple language with a syntax similar to programming languages such as Java. To specify probabilistic behavior, we extended the built-in I/O automata with probabilities and thus obtained probabilistic automata.

Apart from this basic modeling techniques, AF3 features a library concept. Modeling elements, such as (sub-)components or (sub-)functions can be copied into the library. From there they can be instantiated in a model. When changes are applied to a library element, the changes are automatically propagated to all instances of this library element. Hence, the library offers a simple reuse mechanism.

Prism Modelchecker

To perform the availability analysis we use the probabilistic modelchecker Prism (Kwiatkowska et al., 2011). Prism takes as inputs several variants of probabilistic state machines, such as Deterministic Time Markov Models (DTMC) or Markov Decision Processes (MDP). An MDP is a form of probabilistic state-machine that includes local non-determinism. Local non-determinism in this context refers to the situation where for a given state and a given input, there is not one but several probability distributions describing which transition is taken next. The choice, which probability distribution is applied, is taken non-deterministically. A Prism specification is structured into *modules*. Each module defines a set of variables and a set of commands consisting of guards and probabilistic actions. From the set of commands at most one command is executed in each step and from this command one action is selected according to the associated probability distribution. Modules in Prism can be composed in parallel. The standard composition semantics in Prism is composition by interleaving, however, synchronous composition can be achieved by attaching labels to commands that should always synchronize. Transitions and states in a Prism model can be labeled with a number, called *reward* in this context. See Listing 7.1 for an exemplary Prism specification. The example specifies a component that can fail and be repaired. There is a local non-determinism between the two repair commands with different probability distributions. The non-deterministic choice between the two repair commands models uncertainty about the complexity of the fault. Faulty states are annotated with reward 1, good states are annotated with reward 0. The rewards in this case model the costs that are caused by a fault.

For property specification and quantitative queries, Prism supports a range of specification languages. Most relevant here is the language fragment for specifying properties on rewards. Most importantly, Prism allows querying the expected maximal and minimal cumulative reward. This refers to the expected sum of all rewards encountered along an execution path. For example, to obtain the minimal expected cumulative reward in the first 100 steps, the corresponding query would be: `Rmin=?[C<=100]`.

AF3 to Prism Translation

To analyze the AF3 models using Prism, we created an AF3-to-Prism export function integrated into AF3. The export translates the whole AF3 model to an MDP and every AF3 component to

```
mdp

module faultyComponent
  fault:bool init false;
  [fail]  !fault ->  0.01: (fault'=true)  + 0.99: (fault'=false);
  [repair] fault ->  0.1:  (fault'=false) + 0.9:  (fault'=true);
  [repair] fault ->  0.05: (fault'=false) + 0.95: (fault'=true);
endmodule

rewards
  fault:1;
  !fault:0;
endrewards
```

Listing 7.1: Example of a Prism model specifying a component that can fail and be repaired.

a Prism module. All modules are synchronized via a common action label, thus enforcing the time-synchrony of Focus and AF3.

We represent metric functions in AF3 by dedicated AF3 components. The output ports of these special components are translated to reward structures in the Prism model and can therefore be subject to queries regarding for instance their expected value. Hence, availability metric models in our approach are represented in our tool chain by a combination of AF3 modeling elements and statements in the Prism property language on rewards.

Using the above combination of AF3 and Prism we were able to create all models contained in the artifact model described in Chapter 5.

7.4.2 Case Example

Trainguard MT System

The case example in this study is the Siemens product Trainguard MT (TGMT)[2]. It is an automatic train control system for metros, rapid transit, commuter and light rail systems. It is a communication based train control (CBTC) system with bidirectional continuous data communication between the train and the wayside systems. TGMT provides a large number of protection and automation functions for railway operation and uses components on the wayside and on-board the trains.

One purpose of TGMT is to control and protect passenger transfer at platforms. Therefore, TGMT provides a function to control train doors and platform screen doors (PSD), which are installed at the platform and which serve the protection of passengers in metro systems. Figure 7.1 shows a typical platform screen door installation.

For the realization of the door control, TGMT has an interface to the wayside doors and to the train doors on-board the train. In the highest automation mode, all door actions, such as door release, door opening and door closing are performed fully automatic. However, manual operation is still possible. The opening and closing of train doors and PSDs has to be synchronized. Apart from the core functionality regarding the door opening and closing, a number of protective mechanisms are implemented into the system to guarantee passenger safety. The TGMT system must guarantee a high availability of the railway network. Hence, there are strict availability requirements imposed on the system to maintain a continuous railway

[2]http://sie.ag/1aHfP1J

Figure 7.1: An installation of platform screen doors at a railway station.

operation. As an input for our case study Siemens provided some of the original development documents: among them a high-level system requirements specification (59 pages) and a more detailed system architecture specification (299 pages).

Study Scope

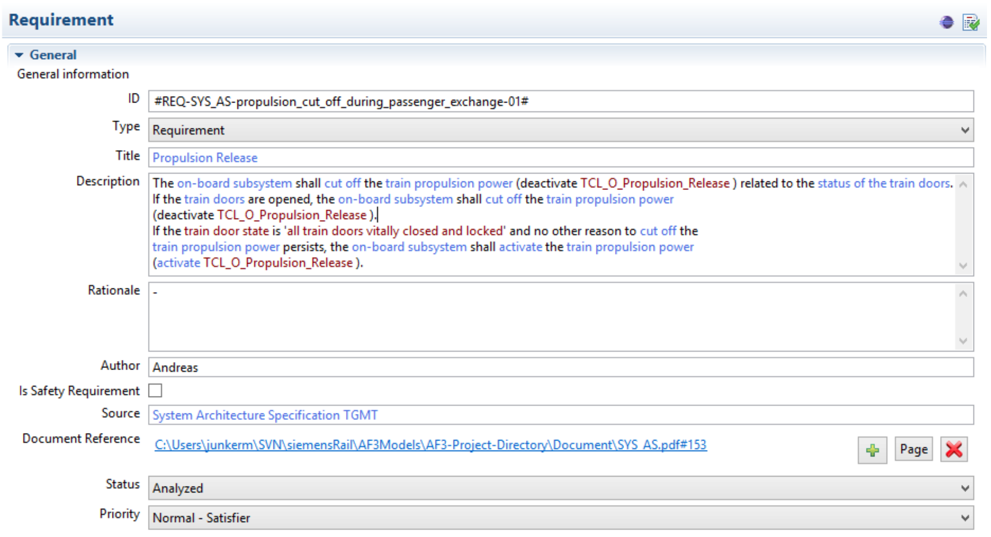
In this study, we did not consider the whole TGMT system, instead we focused on the door controlling functionality of this system. This scope has initially been proposed by Siemens to keep the study at a manageable size. As mentioned before, the TGMT system consists of a subsystem that is deployed directly on the train, called Onboard Control Unit (OBCU) and a wayside subsystem, called Wayside Control Unit (WCU). For this study, we only consider the OBCU part of the case example.

Existing System Model

In a first phase we created a comprehensive system model of the door control functionality of the TGMT system in AF3. We reported on this work in (Böhm et al., 2014). That model forms the basis of this study. Below, we give a brief overview over the TGMT system model.

Functional Requirements Model. For each of the functional requirements from the original documents, the requirements model in AF3 contains an informal description of this requirement in natural language together with meta-data such as an ID, the author, a rationale, etc (Figure 7.2a). For most functional requirements the model furthermore contains a formalization. The formalization consists of a syntactic interface with formal input and output channels for the inputs and outputs that the requirements refers to (Figure 7.2b). We also formalized the system behavior that the requirement describes. In many cases, we specified the behavior using temporal logic patterns (Figure 7.2c). In few cases, description techniques such as automata or message sequence charts (MSCs) are used for the formalization.

Functional Architecture Model. The functional architecture model consists of four high-level functions, listed in Table 7.1. Some of these high-level functions contain sub-functions, which



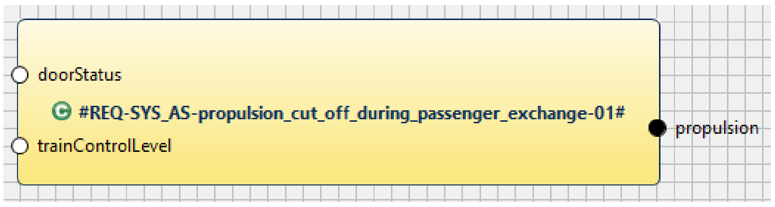
Requirement

General

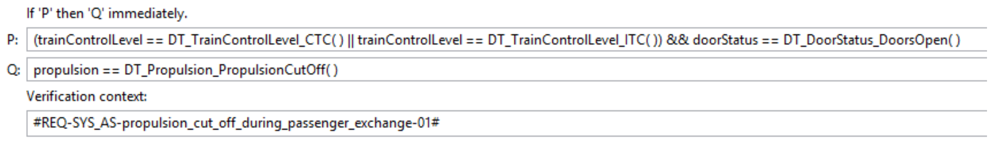
General information

ID	#REQ-SYS_AS-propulsion_cut_off_during_passenger_exchange-01#
Type	Requirement
Title	Propulsion Release
Description	The on-board subsystem shall cut off the train propulsion power (deactivate TCL_O_Propulsion_Release) related to the status of the train doors. If the train doors are opened, the on-board subsystem shall cut off the train propulsion power (deactivate TCL_O_Propulsion_Release). If the train door state is 'all train doors vitally closed and locked' and no other reason to cut off the train propulsion power persists, the on-board subsystem shall activate the train propulsion power (activate TCL_O_Propulsion_Release).
Rationale	-
Author	Andreas
Is Safety Requirement	☐
Source	System Architecture Specification TGMT
Document Reference	C:\Users\junkerm\SVN\siemensRail\AF3Models\AF3-Project-Directory\Document\SYS_AS.pdf#153
Status	Analyzed
Priority	Normal - Satisfier

(a) Informal TGMT requirements as captured in AF3.


doorStatus
#REQ-SYS_AS-propulsion_cut_off_during_passenger_exchange-01#
trainControlLevel
propulsion

(b) Syntactic interface derived from the informal requirement.



If 'P' then 'Q' immediately.

P: (trainControlLevel == DT_TrainControlLevel_CTC( ) || trainControlLevel == DT_TrainControlLevel_ITC( )) && doorStatus == DT_DoorStatus_DoorsOpen( )

Q: propulsion == DT_Propulsion_PropulsionCutOff( )

Verification context:

#REQ-SYS_AS-propulsion_cut_off_during_passenger_exchange-01#

(c) Formal behavior specification with temporal logic patterns.

Figure 7.2: Requirement models in AF3: Informal requirements, formal syntactic interface, and formal behavior specification.

Function	Description
Train Door Control Function	Controlling of releasing, opening, closing, and locking the train doors.
PSD Control Function	Authorizing PSD opening and signaling PSD opening and closing.
Propulsion Function	Deactivating the train propulsion during passenger exchange.
HMI Status Function	Showing door-related information on the driver screen.

Table 7.1: The high-level OBCU functions in the case example.

may again contain sub-sub-functions and so forth. For each leaf function in this function hierarchy, the AF3 model contains a behavior specification. Several of the functions are connected via mode channels. Figure 7.3 shows the functional architecture model on the highest level.

Logical Architecture Model. The primary concern driving the logical architecture of the OBCU subsystem is safety. Accordingly, on the highest level of the logical architecture, there are two main components: ATP, which contains the part of the system that is considered safety relevant, and ATO, which contains the parts not considered safety relevant. Additionally, a third component, ITF is responsible for the communication with the wayside system and provides an interface to the Human-Machine-Interface (HMI). The complete high-level logical architecture of the OBCU is depicted in Figure 7.4. Note that most functions from the functional architecture are realized by more than one component. For example, the train door function is implemented by all three components.

Technical Architecture Model and Deployment. As the technical architecture of the OBCU was not the main focus of the original collaboration, the corresponding model is rather simple. It is quite similar to the logical architecture. It contains one ECU with special hard- and software to perform safety-relevant computations and two further ECUs with standard hard- and software. The ECUs are interconnected by a network. The ATP component from the logical architecture is deployed on the safe ECU, the other two components are deployed on the regular ECUs. As there is a one-to-one correspondence between high-level components and ECUs, we name the ECUs according to the deployed component (ATP_ECU-1, ATO_ECU-1, ITF_ECU-1). Figure 7.5 shows the technical architecture of the OBCU.

Fault Tolerance. The OBCU has the ability to detect a crash of one of its ECUs automatically. For safety reasons, the OBCU shuts down completely in such a case. To continue the train operation, a second OBCU is running as a hot spare. In case of a shutdown of the first OBCU the train transfers the control automatically to this redundant OBCU. In case of a crash of the second OBCU, the emergency shutdown applies as well and the train is brought into a safe state. We adapted the logical architecture as well as the technical architecture to reflect this fault tolerance mechanism. In the technical architecture we added a second set of ECUs

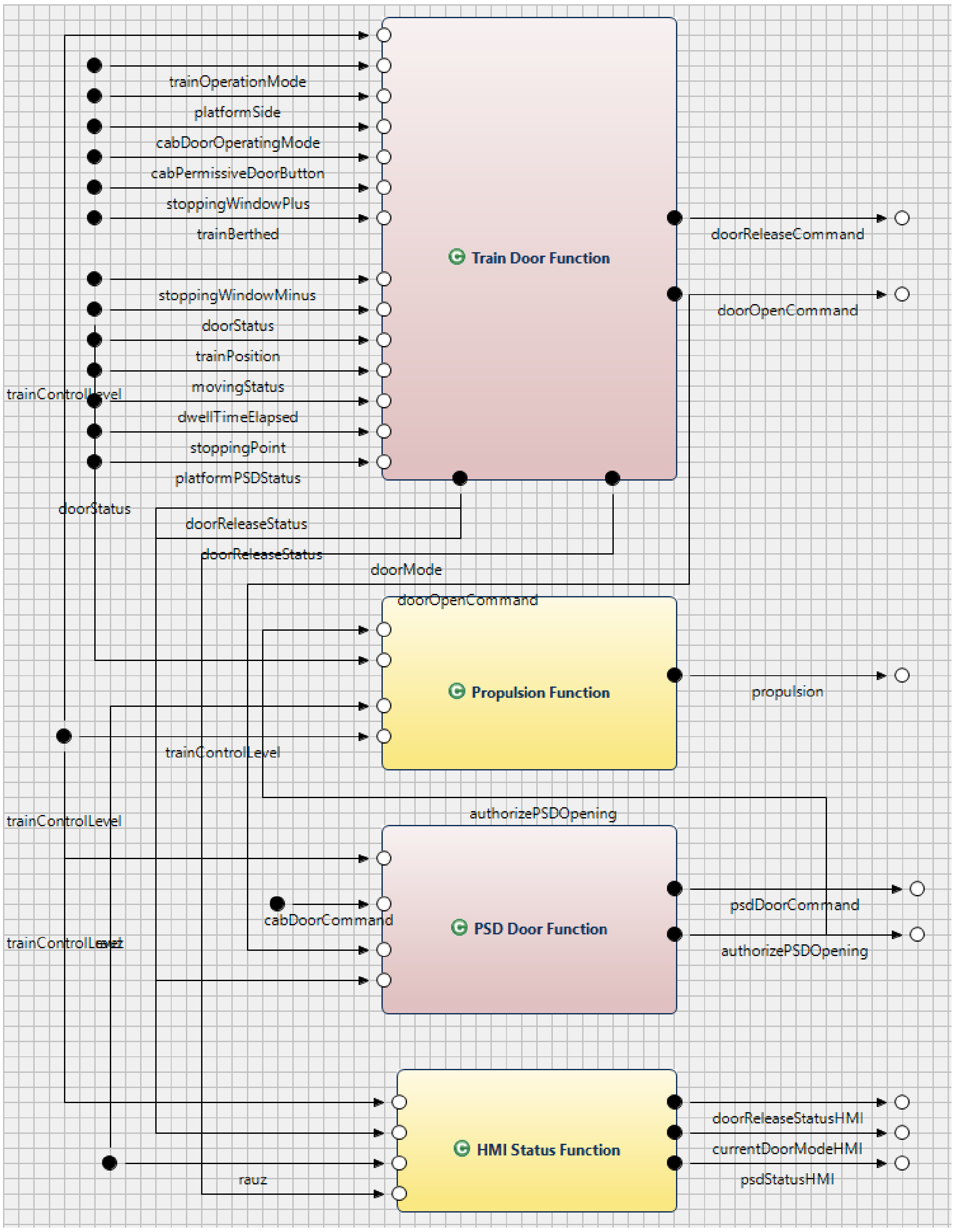


Figure 7.3: The functional architecture model on the highest level showing the four high-level system functions, their interfaces and interconnections via mode channels.

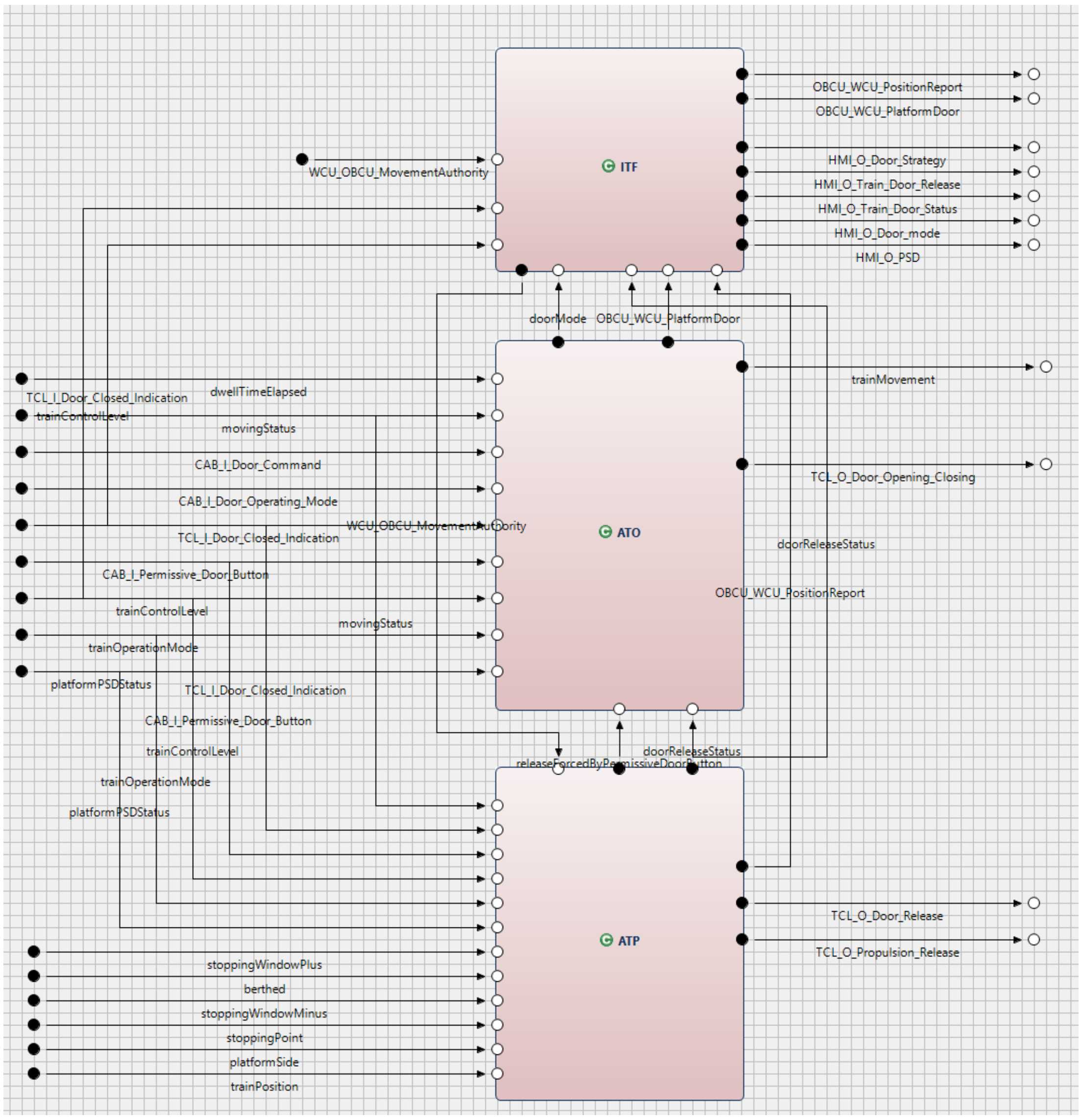


Figure 7.4: The high-level logical architecture of the OBCU.

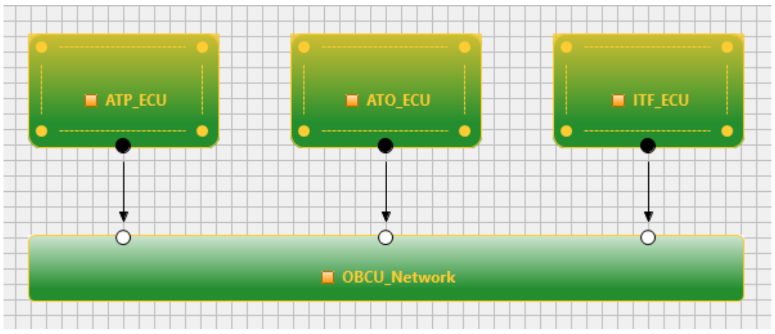


Figure 7.5: Technical architecture model of the OBCU.

(ATP_ECU-2, ATO_ECU-2, ITF_ECU-2). In the logical architecture we duplicated the OBCU component and introduced a switch component that forwards the signals of the second OBCU in case the first one fails. To model the fault detection, we added an additional channel *obcu_fail* to the interface of the OBCU component that signal the crash of the OBCU. Figure 7.6 shows an high-level overview over the adapted logical architecture.

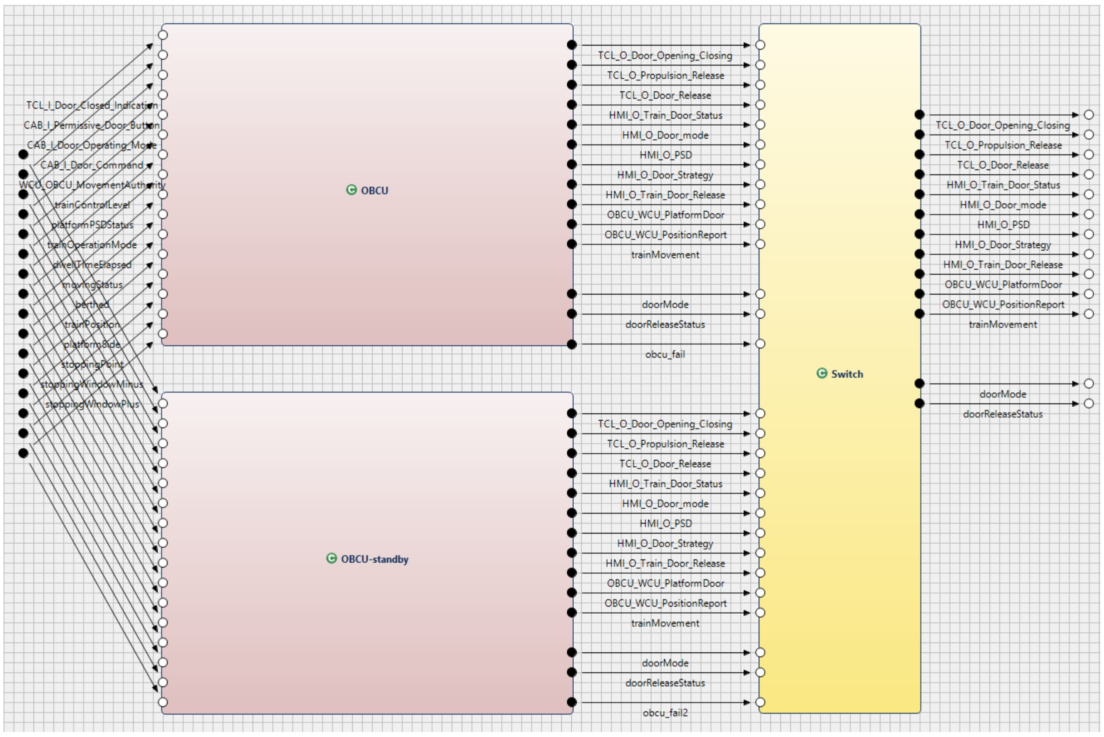


Figure 7.6: Modeling of the two redundant OBCUs with hot standby. The component *Switch* switches between the main OBCU and the hot-standby OBCU in case of a fault signaled by the channel *obcu_fail*

.

Environment Model. The environment model includes the behavior of external systems, such as the interlocking system or the platform automation system (which encompasses the PSD doors), but also the parts of the TGMT system that were not in the scope of the project, such as the functionality to locate the train. It also includes the behavior of a train conductor. The behavior of the environment model is cyclic, modeling the trip between two stations and subsequent passenger exchange.

7.5 Study Execution

In this section we describe the steps we performed as part of the study. Figure 7.7 shows an overview over the study execution. Departing from the existing models described in the previous section, we performed six steps: We first elicited informal availability requirements for the TGMT OBCU. Afterwards we added fault-injection models to the logical architecture model.

Next we created failure definition models and availability metric models. As intermediate steps we created the failure mode list and the aggregation models. We performed an availability analysis for these models aferwards and finally elicited, modelled and analyzed change scenarios. In the following, we describe for each step what we did, what inputs we used and what outputs we created. We distinguish between two types of outputs: The first type are models, such as a failure definition model, the second are availability results and performance measurements from the automated analyses. Most steps in the process used information from the Siemens engineers as inputs. We gathered this information in informal interviews with the Siemens engineers.

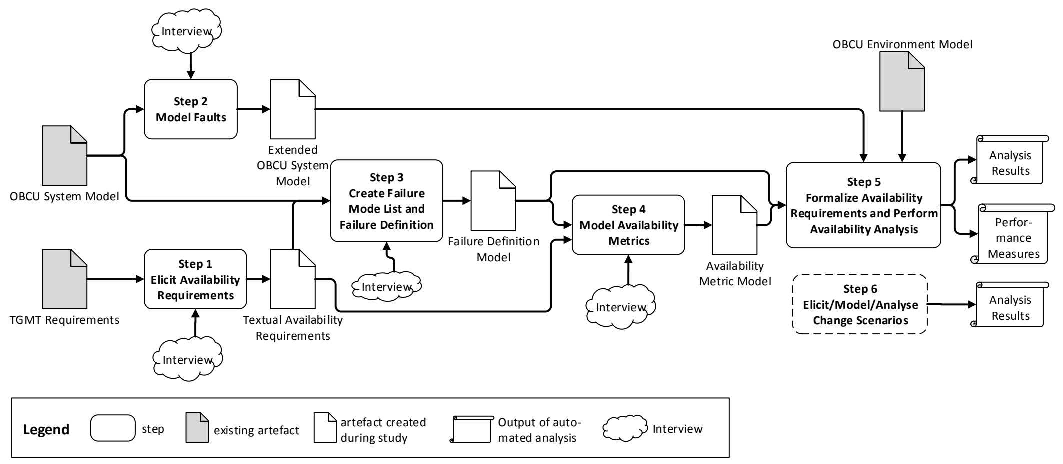

Interview
Step 2 Model Faults
OBCU Environment Model
Extended OBCU System Model
OBCU System Model
Step 3 Create Failure Mode List and Failure Definition
Failure Definition Model
Step 4 Model Availability Metrics
Availability Metric Model
Step 5 Formalize Availability Requirements and Perform Availability Analysis
Analysis Results
Perfor-mance Measures
Step 1 Elicit Availability Requirements
TGMT Requirements
Textual Availability Requirements
Interview
Interview
Interview
Step 6 Elicit/Model/Analyse Change Scenarios
Analysis Results
Legend
step
existing artefact
artefact created during study
Output of automated analysis
Interview

Figure 7.7: Overview over the study execution process. Step 6 is displayed with a dashed line as it is in fact a repetition of steps 3 to 5 with a changed system model.

Step 1: Eliciting Availability Requirements

In this step we first extracted initial informal availability requirements from the requirements documents and afterwards refined them together with the Siemens engineers.

Initial Availability Requirements

In both documents provided by Siemens there is a section "Availability" containing availability requirements. We found that the availability requirements in these documents are partly redundant. The requirements found in the documents can be categorized in three types:

- quantitative availability requirements,
- redundancy requirements, and
- fault tolerance requirements.

The quantitative requirements demand an availability ≥ 99.999% for the whole TGMT system and several sub-systems. However, the documents did neither specify which types of failures should be included into this number nor how this metric should be interpreted. The redundancy requirements describe the hot-standby redundancy architecture of the OBCU and finally the fault-tolerance requirements demand that the system should tolerate certain faults (e.g. of the radio) for a certain amount of time. The latter type of availability requirements is refined by a set of functional requirements according to the tracing information in the documents. For this case study we only considered the first type of availability requirements as it matches with our understanding of quantitative availability requirements. We consider the other types of requirements to be architecture requirements or functional requirements.

Refining the Requirements

To refine the availability requirement we discussed with the Siemens engineers which notion of failure and which availability metric they already use or find appropriate for the system or different system functions. Two general principles that the availability engineers follow are:

- For the evaluation of availability, all functions necessary for the highest automation mode of the TGMT system are considered. The functions we consider in this case study are all part of the highest automation mode and thus subject to the availability requirements.
- The availability engineers evaluate the availability of the OBCU with respect to the operation of the overall TGMT system. This means, from an availability perspective only the failures of the OBCU should be considered that disturb the train operation. In most cases this means train delays.

For the generic (non customer-specific) system the Siemens engineers usually do not describe the function-specific failure modes in more detail but use the above principles to deduce which faults ultimately lead to a system failure. With the availability metric of 99.999% they refer to the steady-state availability.

To summarize, from the initial requirements documents and interviews with the Siemens engineers, we obtained the following informal availability requirements for our case study:

AR1: All functions need to operate with steady-state availability $\geq$ 99.999%.

AR2: A function is considered failed if it is not operating according to its specification and this results in a disturbance of the regular train operation.

Step 2: Modeling Faults

First, we modeled the faults of the main OBCU hardware components ATO_ECU, ATP_ECU, ITF_ECU (for the main and the standby OBCU) using the method described in Section 6.4.3. For each of the hardware components we modeled a complete crash, where inputs are no longer processed and outputs are no longer produced. We obtained fail and repair times from the Siemens engineers: For their analyses, they assume a mean-time-between-failure of 10^4 hours and a mean-time-to-repair of 8 hours, which we adopted. To obtain a per-time-unit fault probability from these values, we made the following assumptions.

- The time-between-failure follows a discrete geometric distribution.
- In the system implementation, the logical time units in AF3 translate to computation cycles with a fixed cycle time of 200ms.

The geometric distribution can be considered the discrete counterpart of an exponential distribution, which is often used for availability analyses in practice (Bracquemond and Gaudoin, 2003). With the assumption of a geometric distribution and the duration of a computation cycle of 200ms, we obtain a fault probability per cycle of about $1.4 \cdot 10^{-8}$ and a repair probability of about $6.9 \cdot 10^{-6}$. We modeled the faulty behavior according to the method proposed in Section 6.4.3. More specific, an ECU fault is modeled in the logical architecture by a post-filter that suppresses any outgoing signal. Figure 7.8 shows an example how such a filter is applied to the ATP component.

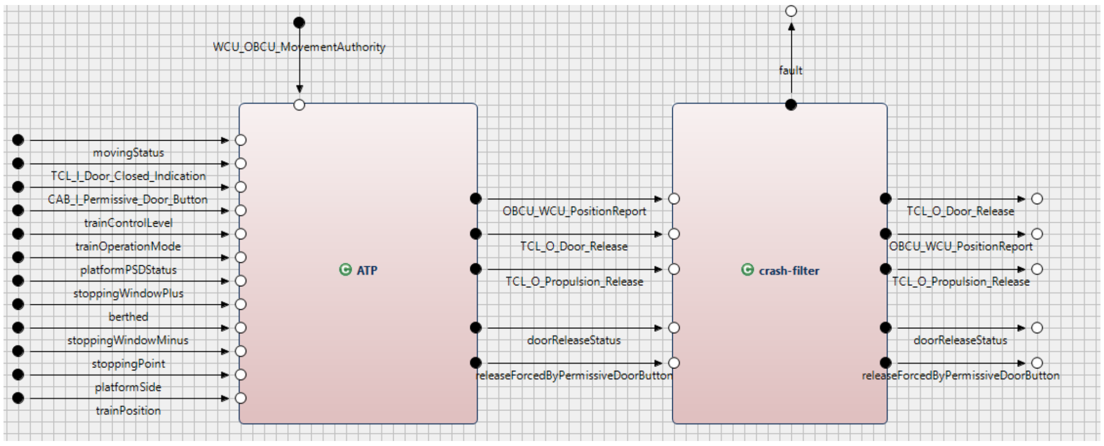


Figure 7.8: Component ATP extended by a post-filter that models the crash failure.

Step 3: Creating Failure Mode List and Failure Definitions Model

In this step we first identified relevant failure modes for the four system functions. Afterwards we created formal failure definition models.

Failure Mode List

In Section 6.4.5, we proposed to use guide words to identify failure mode candidates. We applied this technique to the case study to identify the relevant failure modes for all four system functions. We illustrate the results of the failure mode identification with the PSD function. This function has two output channels. The channel psdDoorCommand carries the signals for opening or closing the doors. The channel authorizePSDOpening is part of the safety functionality that should prevent the PSD doors from being opened accidentally. Hence, prior to opening the doors, an authorization is sent via this channel. For both of the two output channels of the PSD function we obtained failure modes for the guide words Omission, Modification and Late Timing in the discussion with Siemens. The rationales for choosing these failure modes with respect to the psdDoorCommand are:

- The omission of the signal for opening or closing the PSD door delays the train operation as the train cannot depart from the station until the passenger exchange has taken place.
- Similar, if the signals for opening/closing the doors are mixed up, the departure of the train is delayed.

Output Channel	Failure Mode	Description
psdDoorCommand	fm_psdDoorCommand_ommit	The open/close PSD door command is not sent
	fm_psdDoorCommand_mod	Open is sent instead of close or vice versa
	fm_psdDoorCommand_del	The open/close PSD door command is sent with delay
authorizePSDOpening	fm_authorizePsdOpening_ommit	The authorize PSD door command is not sent
	fm_authorizePsdOpening_mod	Not authorize is sent instead of authorize
	fm_authorizePsdOpening_del	The authorize PSD door command is sent with delay

Table 7.2: Failure modes for the PSD system function.

- Finally, a delay in opening or closing the PSD doors leads to delay in train operation.

Regarding the channel authorizePSDOpening the rationales are:

- Only the omission of an authorization message is considered. When the authorization for opening the PSD doors is not sent prior to opening the doors, the opening will not happen and the train operation is delayed. The case of an omitted not-authorize message is *not* considered as this is a safety issue but does not prevent the train from departing the platform.
- Similar, only the accidental sending of a not-authorize message instead of an authorize message is considered a failure as only this case leads to a train delay.
- Finally, a delay of the authorize message leads to delay in train operation and is therefore considered a failure whereas the delay of a not-authorize message is neglected.

Table 7.2 shows an overview over the identified failure modes for the PSD function. For all of the failure modes of the PSD function we only employed one level of severity.

Failure Definition Models

Based on the identified failure modes we created failure definition models. For this step we followed the proceeding and pattern as outlined in Section 6.4.6. We used the failure definition model template and created the failure definition models based on the original function specification and filters from our building blocks. We instantiated the failure definition model pattern for each system function. Although this could be automated by extending AF3, in our case study we did this manually. Recall from Section 6.4.6 that the necessary ingredients of this pattern are the function specification, a number of deviation models and a comparator. Figure 7.9 shows the failure definition model for the PSD door function. Although the graphical syntax for specifying deviation models from the previous chapters is not available in AF3, the pattern structure with the filter chain is clearly visible in the center of the diagram. Each of the filters is an instantiation of a basic building block. The comparator is on the right side of the diagram.

Step 4: Modeling Availability Metrics

In this step, we first decided on suitable availability metric models from the basic building. As described in Section 6.4.8 creating a failure aggregation model is often necessary to fit the

++

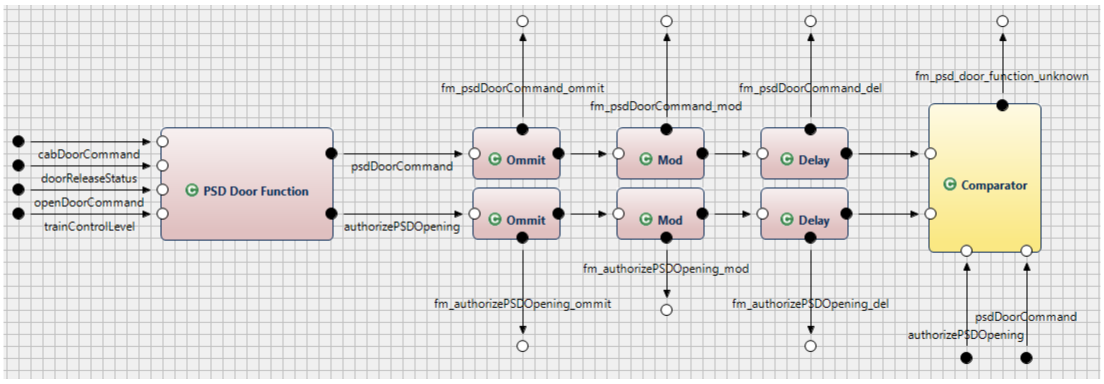


Figure 7.9: Failure definition model of the PSD door function. It is structured according to the template for failure definitions and uses the basic building blocks for the deviation filters. It comprises the function specification (leftmost box), the deviation filters (six middle boxes) and a comparator (right box).

failure definition and the metric model together. We created the aggregation model as part of this step.

Availability Metric Model

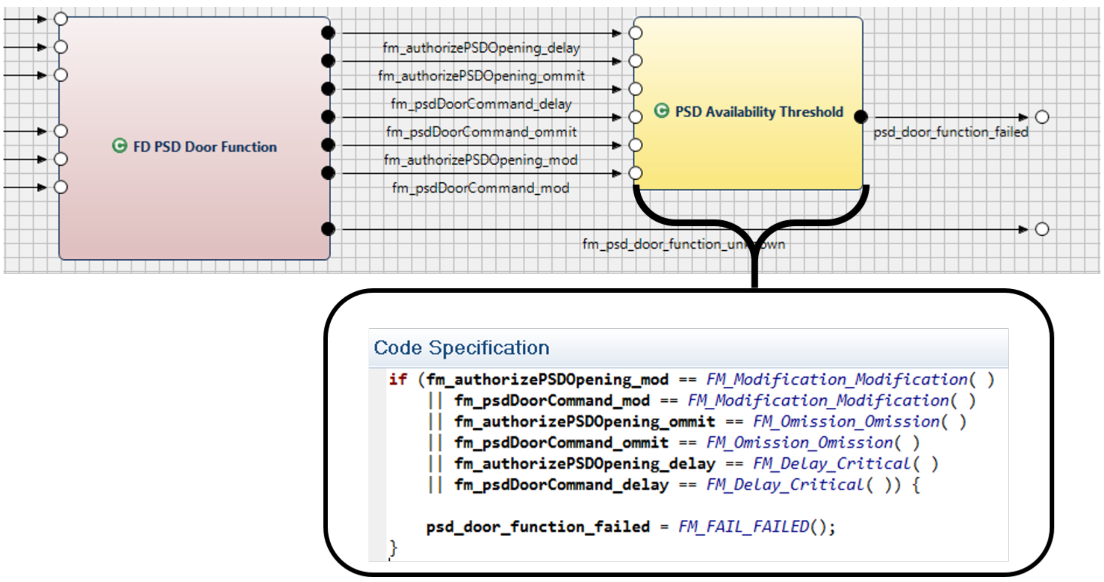
Steady-state availability should be used as an availability metric according to our informal requirements. However, we found that there is no translation of steady-state availability in our tooling, as in Prism the steady-state rewards operator is currently not supported. Hence, we were not able to use the basic building block for steady-state availability. Therefore, for this case study, we replaced the steady-state availability with the interval availability during 24 hours for which we have a representation in Prism and can use the basic building block for interval availability.

Aggregation Model

Choosing to employ the basic building block for interval availability made it necessary to define an aggregation model to aggregate the failure modes of the failure definition models to a single failure mode. Figure 7.10 shows the failure definition for the PSD door system function (at the left) and the according aggregator (at the right).

Step 5: Performing Availability Analysis

In this step we performed availability analyses based on the created models to verify the availability requirements and to determine the impact of the MTTF and MTTR parameters of the single ECUs on the availability. We evaluated the efficiency of the tooling by measuring analysis time and use of memory. We plausibilized the results of the analyses by comparing it with the output of a commercial analysis tool.


FD PSD Door Function
PSD Availability Threshold
fm_authorizePSDOpening_delay
fm_authorizePSDOpening_ommit
fm_psdDoorCommand_delay
fm_psdDoorCommand_ommit
fm_authorizePSDOpening_mod
fm_psdDoorCommand_mod
psd_door_function_failed
fm_psd_door_function_unknown
Code Specification

```
if (fm_authorizePSDOpening_mod == FM_Modification_Modification( )
    || fm_psdDoorCommand_mod == FM_Modification_Modification( )
    || fm_authorizePSDOpening_ommit == FM_Omission_Omission( )
    || fm_psdDoorCommand_ommit == FM_Omission_Omission( )
    || fm_authorizePSDOpening_delay == FM_Delay_Critical( )
    || fm_psdDoorCommand_delay == FM_Delay_Critical( )) {

    psd_door_function_failed = FM_FAIL_FAILED();
}
```

Figure 7.10: The failure definition for the PSD door function (left box) is complemented by an aggregator (right box). The aggregation is defined using an AF3 code specification (lower box). The aggregator specifies that the function is considered failed (`psd_door_function_failed = FM_FAIL_FAILED()`) if any of the single failure modes is present (`if`-clause).

Availability Analysis

We performed two types of analyses:

- *Requirements Verification:* We determined availability estimates using our tooling and thus verified whether the system, as we modeled it, does fulfill the availability requirements and

- *Sensitivity Analysis:* We performed a sensitivity analysis indicating how varying the parameters MTTF and MTTR of the different ECUs influences the availability of the system functions. This analysis further unveils whether there are components that are especially critical for the availability. We chose MTTF values of 3000h, 4000h, and 5000h and MTTR values 7h, 8h, and 9h. We only varied one parameter for one ECU at a time. All other parameters were left at the defaults 4000h for MTTF and 8h for MTTR.

We performed all our experiments on a computing node with an AMD Opteron processor at 2.6 GHz and 2 GB of RAM. Note that the Prism modelchecker works mostly sequential. However, we used several computing nodes to run several single analyses (e.g. for different functions or with different parameters) in parallel. To evaluate the feasibility of the analysis in terms of computing resources, we measured the run-time and memory consumption.

Plausibilization of the Results

We took two measures to assess the plausibility of the analysis results. First, we manually created an analysis model in form of a reliability block diagram (RBD) reflecting the fault tolerance mechanism of the OBCU. This was possible for the original model as the complete shutdown of the OBCU does not give rise to a complex failure behavior and hence the situation could be modeled with an RBD. In this RBD we annotated the same MTTF and MTTR parameters as in our model and specified an exponential failure distribution for both, time-to-failure and time-to-repair. However, the resulting RBD does not differentiate between different failure modes. Hence, small differences between the results of the RBD analysis and our analysis were expected. We used the commercial reliability modeling and analysis tool ReliaSoft BlockSim[3] to perform a second availability analysis independent of our models. We compared the results to assess if the values we determined by our method are plausible.

Step 6: Analysis of Change Scenarios

A specific challenge for Siemens is to adapt the availability analysis to changing systems and changing requirements. This is especially important in case the system is adapted for a specific customer. We posed RQ3 to assess the ability of our approach to cope with such a situation. To answer this question, we first collected possible change scenarios. This includes changes in the functionality and changes in the (logical and technical) architecture. In a second step we changed the system models according to the collected change scenarios. We then adapted the availability analysis models and performed an availability analysis or a sensitivity analysis based on the changed models.

Elicitation of Change Scenarios

In interviews with the Siemens engineers, we elicited a number of possible variation points in the system. We obtained three types of variation points:

Variation in the system configuration: Certain aspects of the system behavior are described via parameters. Examples are the rail network profile or characteristic values for acceleration and deceleration. These configuration parameters vary between different customers.

Variation in the system functionality: Not all customer-specific instances of the TGMT system use all of the system functions. For example, the PSD functionality is left out by some customers.

Variation in the system architecture: Due to legacy systems already deployed at the customer and the need to integrate into a given architecture or due to the evolution of the system, the (logical or technical) architecture of a concrete system instance may deviate from the architecture of the generic system.

From these variation points, we only considered the second and third in our study. Configuration parameters are barely considered in the AF3 model of the TGMT, as they hardly relate to the door functionality. Hence, we obtained two types of change scenarios for the case study. For the first of these change types, we analyze one concrete change scenario (labeled S1), for the second, we present two concrete change scenarios (labeled S2a and S2b).

S1: Removal of the PSD door function: The PSD door function is only employed in rail systems with installed platform screen doors. If this is not the case and thus the PSD door

[3]http://www.reliasoft.com/

function is deactivated for the customer specific installation, this should be reflected in the availability analysis. Hence, in this scenario, we consider a system without the PSD functionality.

S2a: Removal of the ITF component: In the original logical architecture, there are three top-level components, relating to three ECUs in the technical architecture: ITF, ATO and ATP. The ITF is responsible for the communication between the OBCU and the wayside subsystem as well as for the communication with the train's HMI. In this scenario, we consider a situation where the ITF component as well as the ITF_ECU hardware is removed and all of its responsibilities are delegated to the ATO component and the ATO_ECU hardware instead.

S2b: Change of Fault Tolerance Mechanism: We inspect a second scenario where the architecture changes. In this case, we investigate a change in the fault tolerance mechanism. In the original model, the complete OBCU is shut down in case of a failure in one of its components. We modify this behavior as follows: Only the first OBCU performs the emergency shutdown in case of a component defect. The second OBCU does not perform the shutdown but instead works with the remaining components to provide the functionality as long as possible.

Modeling of Change Scenarios

Change scenario S1. To realize change scenario S1, we removed all components from the system model that exclusively relate to the PSD functionality and hence do not contribute to a different function. We further removed all PSD related behavior from the remaining components. Finally we removed all channels from the interface of the system model that relate to the PSD functionality. To adapt our availability analysis model to the new situation we removed the failure definition and metric models for the PSD function.

Change scenario S2a. We realized changed scenario S2a in the system model by removing the ECU_ITF in the technical architecture as well as the top-level component ITF in the logical architecture and integrating all of its sub-components into the component ATO. See Figure 7.11 for the modified architecture and compare this with the original architecture shown in Figure 7.4. As the black-box interface of the system was not affected by these changes, we did not need to modify the analysis model at all.

Change scenario S2b. The emergency shutdown behavior is modeled in the dedicated Switch component (see Figure 7.6). To realize the change scenario S2b in the system model we therefore modified the switching behavior to prevent the shutdown of the second OBCU. Again, the black-box interface of the system does not change. Therefore, we did not need to adapt the failure definitions and availability metric models.

Analysis of Change Scenarios

Finally, we performed automated requirements verification and sensitivity analyses with the modified models. The setting of the analyses is the same as for the unmodified models.

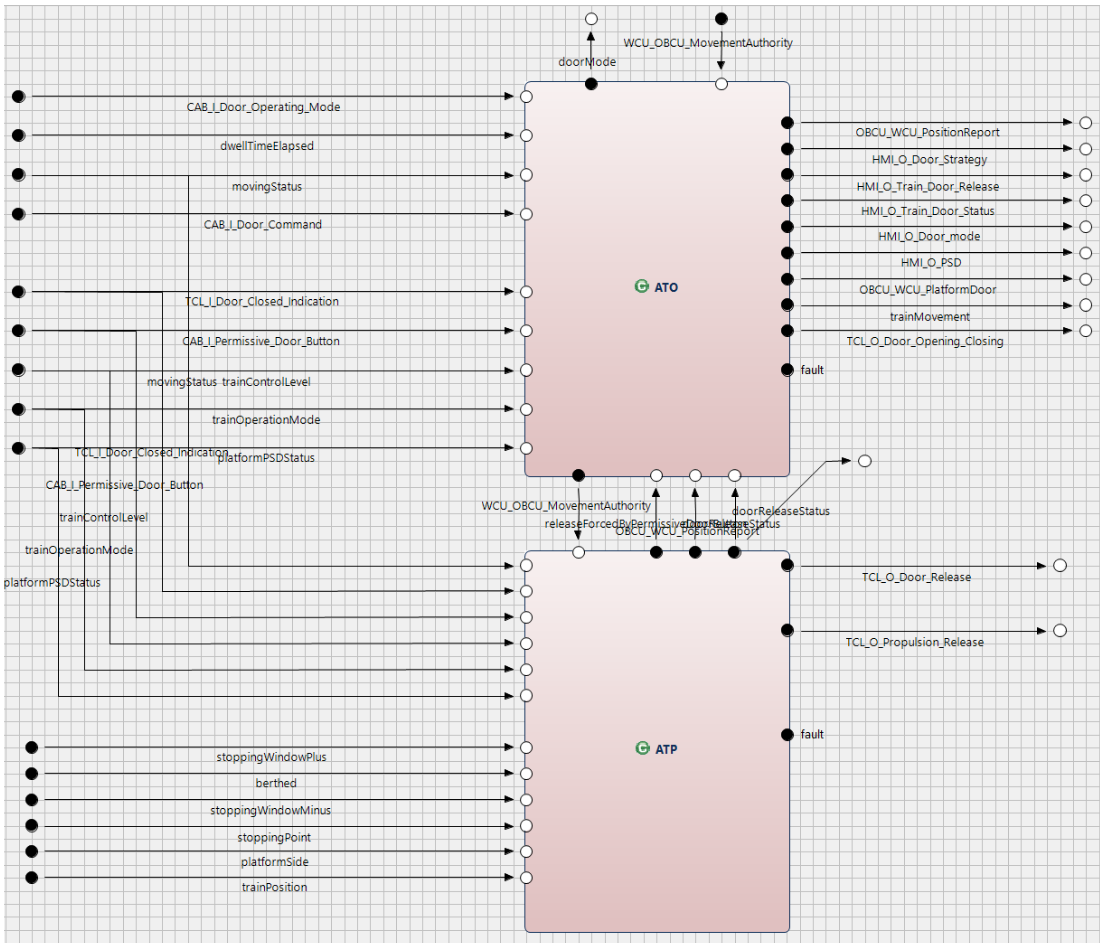


Figure 7.11: Modified logical architecture for change scenario S2a. The ITF component has been removed and all its responsibilities have been moved to the ATO component.

7.6 Analysis Results

In this section, we present the results of performing the availability analyses on the original model as well as on the models obtained from applying the change scenarios.

7.6.1 Analysis of the Original Model

Verification of Availability Requirements

In a first step we verified the availability requirements associated with all system functions. For all functions we determined similar availability results of ≈ 99.99805%. In fact, the results for the different functions varied only in the 8th decimal (in the percentage representation). Hence, our analyses show that all functions fall slightly below their availability requirements (99.99805% instead of 99.999%). We therefore could not verify that the availability requirements are met with the given architecture and parameters (MTTF=4000h, MTTR=8h).

To plausibilize this result, we compared it with the availability estimation from the commercial tool Reliasoft Blocksim. With this method we also obtained an availability estimate of 99.998%, which suggests that we obtained plausible results through our method.

Sensitivity Analysis

We varied the MTTF and MTTR parameters for each of the ECUs and evaluated the impact of this on the availability of all functions. We found that the impacts of varying the parameters are almost identical for each of the functions. This means that when we vary, for example, the MTTF of ATO_ECU-1, this has the same effect on the availability of the propulsion function as on the HMI status function. Furthermore, for all functions the impact of varying the same parameter on different ECUs are the same. This means that when we vary the MTTF of ATO_ECU-1 this has the same effect on the availability of all functions as when we vary the MTTF of ATP_ECU-1 in the same way. We can hence conclude that all ECUs are equally important for the availability of all functions. The reason for this is the shutdown behavior implemented in the OBCU. As a fault in any ECU leads to an immediate complete shutdown of the whole ECU (which affects all functions) all components are equally important.

Due to the two observations above, we only present the detailed results for the sensitivity analysis of the propulsion function with respect to the ECU ATO_ECU-1. One can observe that when increasing the MTTF by 1000h, the availability also increases in the order of magnitude of 10^{-4}. When increasing the MTTR by 1h the availability decreases in the order of magnitude of 10^{-5}. However, our results also show that the availability does not increase (respectively decrease) linear with an increase of the MTTF and MTTR. The numeric results and a visualization of the sensitivity analysis for the propulsion function and ATO_ECU-1 are shown in Figure 7.12. The row labeled with Δ in the enclosed table shows the change of availability when varying the parameters.

7.6.2 Analysis of the Change Scenarios

Change scenario S1: Removal of the PSD Function

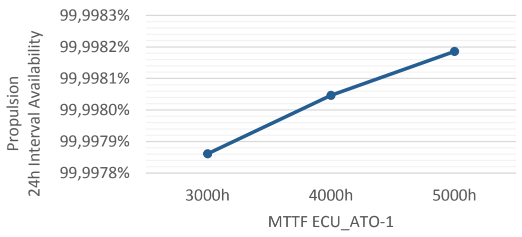
We determined that the availability of the remaining functions is not influenced by the removal of the PSD door function: For all remaining functions the availability values for the system with the reduced functionality is the same as for the original system.

Change scenario S2a

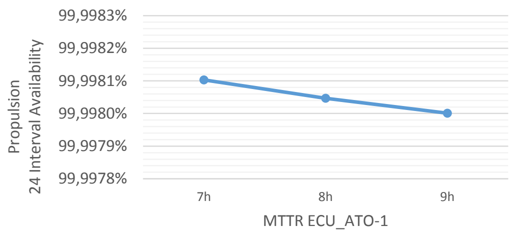
In case of the second change scenario, the availability results for the modified architecture deviate from the ones of the original architecture. The availability increased compared with the values for the original architecture. Instead of an availability of 99,99805% that was achieved by the original architecture, we determined a value of 99,99913% for the changed architecture (for all functions). This is especially interesting as it is above the desired availability 99.999% demanded by the availability requirements. The increase results from the removal of the ITF_ECU as a source of faults. As we assume that the mean-time-to-failure of the ATO_ECU with its now increased workload does not change, the probability of shutting down an OBCU is decreased. Therefore, the overall availability increases.

	ATO-1 MTTF			ATO-1 MTTR		
	3000h	4000h	5000h	7h	8h	9h
	99.99786%	99.99804%	99.99818%	99.99810%	99.99804%	99.99800%
Δ	-	$1.8 \cdot 10^{-4}$	$1.4 \cdot 10^{-4}$	-	$-6,0 \cdot 10^{-5}$	$-4.0 \cdot 10^{-5}$

(a) Availability (in %) of the propulsion function with varying parameters of ECU_ATO-1 MTTF and ECU_ATO-1 MTTR. The row prefixed with Δ shows the differences between the above availability data-point and the data-point one column to the left.



(b) Availability of the propulsion function with varying MTTF of the ECU_ATO-1



(c) Availability of the propulsion function with varying MTTR of the ECU_ATO-1

Figure 7.12: Results of the sensitivity analysis for the propulsion function and varying parameters for the MTTF and MTTR of ECU_ATO-1. The table in (a) shows the plain numbers, while the graphs in (b) and (c) visualize the sensitivity with respect to MTTF and MTTR.

Change scenario S2b

For this scenario the availability results also differ from the original ones. As before, the availability with respect to the changed architecture is higher compared with the original architecture. For example, the availability of the propulsion function in case of the modified architecture with the default parameter values is 99.99934%, compared with 99.99805% in the original architecture and hence above the threshold given by the availability requirements. See Figure 7.13 for a comparison of the availability of the propulsion function in the different architecture alternatives.

The results of the sensitivity analysis show that varying the MTTF and MTTR parameters of an ECU of the second OBCU now has a different impact on the availability of different functions. Consider Figure 7.14 that shows the results of a sensitivity analysis for the propulsion function with respect to the MTTF and MTTR parameters of the ECUs ATO_ECU-1, ATO_ECU-2 and ATP_ECU-2 . Increasing the MTTF of ATP_ECU-2 from 3000h to 5000h (dark blue line in Figure 7.14a) strongly affects the propulsion function. Increasing the MTTF of ATO_ECU-1 (light blue line) also has a small effect. Finally, increasing the MTTF of ITF_ECU-2 (grey line) has no effect at all. Varying the repair time creates a similar picture (see Figure 7.14b). Hence, the ATP_ECU-2 has the highest importance for the availability of the propulsion function.

The reason for this behavior is the following: As a fault of a single component of the second OBCU does no longer cause a complete shutdown and therefore some functions continue to work at least partially, the overall availability increases. As the components involved in the propulsion function are deployed on the ATP_ECU while the components implementing the HMI function are mostly deployed on the ATO_ECU and ITF_ECU, the different impact of

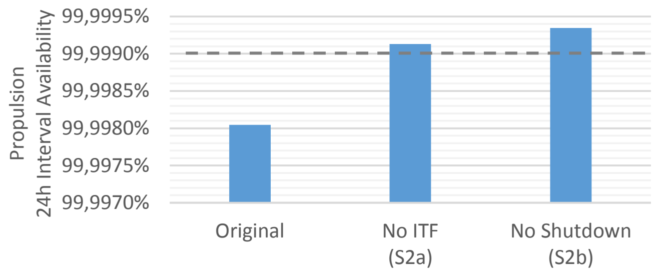
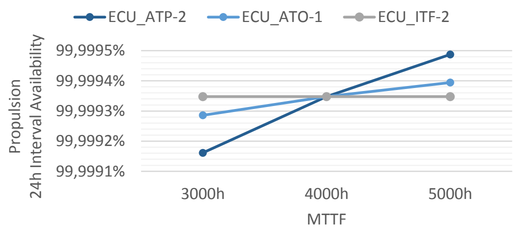
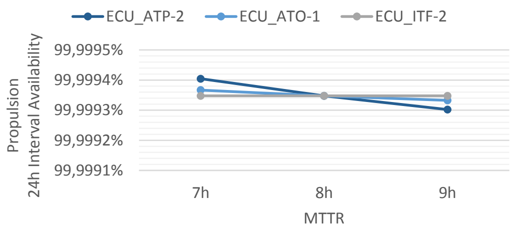


Figure 7.13: Comparison of the availability of the propulsion function with the default parameters in the three different architecture alternatives. The dashed grey line visualizes the availability threshold of 99.999% given by the requirements.

the ECU parameters on the functions can be explained. Note that this architecture may lead to behavior that is unwanted from a safety perspective, which we however do not consider here.



(a) Sensitivity of the propulsion function availability when varying the MTTF of ECU_ATO-1, ECU_ITF-2 and ECU_ATP2.



(b) Sensitivity of the propulsion function availability when varying the MTTR of ECU_ATO-1, ECU_ITF-2 and ECU_ATP2.

Figure 7.14: Sensitivity analysis results for change scenario S2b (no emergency shutdown of the second OBCU) for the propulsion function.

7.6.3 Analysis Performance

The performance parameters are noted in Table 7.3. The size of resulting analysis models varied between 10^5 and 10^6 states depending on the function that we analyzed. The analysis turned out to be costly in terms of analysis time (between 5h and 50h), but only showed a moderate consumption of memory (below 1GB). The latter made it possible to run many analysis instances in parallel, which greatly sped up the sensitivity analysis. We noticed in side experiments that the analysis time is greatly influenced by the interval length considered in the interval availability metric model. In fact, interval length and analysis time are almost proportional. This effect is due to the iterative solution method the model-checker employs for the problem.

7.7 Discussion of the Research Questions

In this section we answer our initial research questions by reflecting on the models and analysis results we obtained during the study execution. We further incorporate feedback by Siemens

Analyzed Function	**States**	**Analysis Time**	**Memory**
Propulsion Function	122,096	5.1 h	383 MB
HMI Function	122,096	4.8 h	362 MB
PSD Function	976,712	55.4 h	691 MB
Train Door Function	488,360	23.5 h	471 MB

Table 7.3: Performance of the availability analysis in terms of time and memory consumption.

into the discussion.

7.7.1 RQ1: Modeling Adequacy

The results show that it was possible to capture the availability requirements of the TGMT system by instantiating our artifact model and by following the steps outlined in Chapter 6. However, the availability requirements were not very complex. The according availability metric could be captured with a rather simple building block.

According to the feedback by Siemens the failure modes elicited through the guide words are considered the relevant failure modes of the system. However, the found failure modes related only to the guide words OMISSION, LATE TIMING and MODIFICATION. We did not identify failure modes based on the guide words EARLY TIMING and INSERTION. The reasons for this are:

- Early timing is not a problem, at least in the part of the system that we considered. Usually all reactions need to be performed as soon as possible when a certain input configuration is present.

- All the signals have been modeled as state signals, which are supposed to be present all the time. Hence, an insertion of a message cannot be distinguished from a modification and, therefore, insertion is not considered.

Applying the failure definition pattern we could create all failure definitions. In the failure definitions we reused the function specifications. To fill the failure definition pattern, we could further apply the filters defined in the basic building blocks.

We also experienced a few limitations when creating the models. Most importantly, we could not capture the original availability requirements that relate to availability in the long run (steady-state availability). This limitation, however, was due to the tooling and not a conceptual issue. As mentioned above, we could not represent the long-run availability in the PRISM modelchecker and, therefore, we used interval availability instead.

In the final discussion with Siemens, one issue emerged concerning function interaction. In case a failure propagates via a mode channel from one function to another, we attribute such a failure to the function where the failure originated. However, the Siemens engineers remarked that we could instead attribute this to the functions where the failure surfaces at the system interface.

In general, we found that the TGMT system fits very well to our approach. However, the used data-types are mostly simple (such as enumerations and primitive data types) and the behavior of the system is to a large extent stateless. Both allows for rather simple failure modes. Nevertheless, we conclude that the case study suggests that our approach provides suitable modeling techniques and a supportive method to capture the availability requirements of a system such as the TGMT system.

7.7.2 RQ2: Analysis Feasibility

The study demonstrates an automated analysis of the system's availability properties based on our artifact model. Both, the verification of the availability requirements and sensitivity analyses are possible. The availability results are useful: The analysis results showed that the availability requirements are slightly missed in the original model. The requirements are, however, fulfilled by the architectures in the change scenarios S2a and S2b. The sensitivity analysis revealed that all ECUs have equal impact on the availability of all functions in the original model but different impact in scenario S2b. The analysis results are furthermore plausible: We plausibilized the results for the original model with a commercial analysis tool based on an RBD.

Our results with respect to the computational feasibility of the analyses are ambivalent. Only a moderate amount of memory was necessary to perform the analyses. However, with the present tooling, the analyses consume a lot of time (between 5 hours and 2 days). The long analysis time is partly due to the size of the models (up to 1 million states). Nevertheless, when taking into account that no further manual analysis tasks need to be performed, such as performing an FMEA or creating dedicated architecture models, the time consumption by the analysis might be acceptable, even without further tool improvements. A different question is how far our method scales with further growing model sizes and more complex failures. As a part of our approach is to analyze the availability per system function we at least partly avoid the state explosion problem, as we only need to include that part of the system that relates to the system function under analysis. However, the more the functions are distributed over the system, the smaller this advantage is. Nevertheless, as we worked with a case example from an industrial context and of a considerable size we argue that there are situations where the automated analysis can be feasibly applied.

7.7.3 RQ3: Flexibility

We investigated three change scenarios in the context of the TGMT OBCU. In all cases the adaptions that we needed to make in the availability analysis model were only trivial and we could reuse almost the whole analysis model. Only in the case of a removed function, we also had to remove the corresponding failure definitions and availability metric models. With the changed analysis models, we could again evaluate the availability and quantify the impact that the changes had on the availability.

An especially interesting result is the availability analysis of change scenario S2b (the prevention of an emergency shutdown in the second OBCU). Here, we could show that a subtle change in the behavior of a system component (in this case, the switching component) can have an impact on the availability of the system. Using a classical approach with an explicit analysis model, such as an RBD based method, this change would have to be analyzed and its effect modeled in the RBD. In our case the changes were immediately reflected in the analysis results without further (error-prone) changes in the analysis models.

Based on the above we conclude that our artifact model and the method allows to create robust analysis models. Even major changes to the system architecture and functionality are reflected by only small changes in the analysis models.

7.8 Threats to Validity

In this section we discuss issues that threaten the validity of the case study results. We distinguish between threats to the construct validity and threats to the external validity. As we did not investigate any cause-effect relationship we do not discuss threats to the internal validity (which is the type of validity concerned with the correctness of cause-effect relationships).

7.8.1 Construct Validity

Construct validity relates to the question if we correctly operationalized the phenomena we are interested in. To answer RQ1, we captured modeling adequacy qualitatively by evaluating whether we could model the availability requirements we obtained from the documents and interviews. The main threat here is that we did not acquire a representative set of availability requirements and failure modes. The selection of requirements could be flawed because the documented requirements are not complete. It might be furthermore possible that the requirements and failure modes that we obtained are influenced by the specific way that Siemens currently performs its RAM analyses or even that we had our approach in mind when selecting the availability requirements. Due to these influences we might have modeled wrong requirements and thus our results are flawed. We tried to mitigate this threat by not only relying on the documents but also doing several interviews with the Siemens engineers and discussing the requirements and the failure modes. However, we worked mostly with engineers responsible for the generic system and not for the customer specific systems. A second threat is that the models do not actually capture the requirements, e.g. because we misinterpreted the requirements or due to misunderstandings in the interviews with the engineers. We tried to mitigate this threat by discussing intermediate models with the engineers. Generally, all modeling was done by us and not by engineers. Therefore we can only assess the ease of modeling from our perspective.

7.8.2 External Validity

External validity relates to the generalizability of our results. The main threat here is caused by the minimal sample size of only one system in one specific domain. Hence, we cannot generalize from the results of this case study to a wider population. This threat can only be dealt with by repeating the case study on more systems from different domains. Similar, it might be possible that we picked a part of the TGMT system that is not representative for the whole TGMT system and thus the results do not even generalize to the whole TGMT system. We dealt with this threat by letting the Siemens engineers choose the part of the system that should be subject to the study based on their knowledge of the system. Furthermore, the requirements documents that we received from Siemens covered the whole system. When scanning through them, we got the impression that the rest of the system is not fundamentally different from the part we covered.

7.9 Conclusion

In the last section we summarize this chapter. Specifically we highlight the benefits as well as the limitations that the presented case study reveals.

7.9.1 Summary

We presented a case study to evaluate the modeling and analysis approach for availability developed in the preceding chapters. The case example was the industrial train control system TGMT developed by Siemens. To evaluate our approach, we posed three research questions: With RQ 1, we assessed the adequacy of the modeling artifacts and the modeling method, with RQ 2, we investigated the feasibility of automated analyses techniques based on our approach, and finally, with RQ 3, we evaluated if our approach realized the claimed benefits regarding reuse of analysis models and robustness to changes in functionality and architecture.

We based the case study on an initial model of the TGMT system developed in a collaboration project with Siemens. We extended this model to incorporate availability relevant aspects such as fault and fault tolerance. In order to perform the modeling and analysis tasks we extended the modeling tool AutoFocus 3 in various ways, especially we integrated the probabilistic model-checker Prism. Based on requirements documents and a series of interviews with engineers, we developed failure definition models and availability metric models. We also performed availability analyses to verify the initial requirements and assess the impact of model and parameter changes on the system availability.

7.9.2 Benefits

We found that our approach could be applied in the given context. We were able to create all necessary models. Only in one case we were unable to faithfully model the requirements. This case, however, was mostly due to tool issues and not a conceptual problem. Using the modelchecker we could obtain useful availability results. We further demonstrated that model changes can be handled without large changes in the analysis models by investigating three concrete change scenarios. In one of this change scenarios we showed how subtle changes in the system behavior can influence the system's availability and how our approach can handle such a situation.

7.9.3 Limitations

A major limitation of the automated analysis is the long time span needed to perform the analysis. However, there are several opportunities for improving the current tool chain, for example:

- Much of the consumed time the modelchecker spends on iterations to refine its solution. The number of iterations directly relates to the length of the interval of the interval availability metric. Starting with a smaller interval and gradually increasing this interval until a saturation point is reached could speed up the analysis greatly.
- We used exact modelchecking in this case study. A different approach would be to use statistical simulation. The main challenge in statistical simulation is dealing with the non-determinism. Often, non-determinism in such a case is handled by replacing it with a uniform probability distribution. However, this is not an option in our case, as we are interested in the extreme values for all possible resolutions of the non-determinism. However, approaches for dealing with non-determinism in a sound way have been proposed recently (Brázdil et al., 2014; Hartmanns and Timmer, 2015; Henriques et al., 2012). Applying these techniques for our case could speed up the analysis considerably.

A further limitation we experienced is the restricted set of availability metrics that we could model. However, this is a tooling issue due to the used model checker and not a conceptual issue.

Chapter 8

Conclusions and Outlook

In this chapter, we summarize the contributions presented in this thesis and outline possible future research directions.

8.1 Conclusions

In this thesis, we considered the model-based specification and analysis of the availability of software intensive systems. In an analysis of the state of the practice and the state of the art we found that there are open problems which are not targeted by current research. The two problems that we consider in this thesis are:

- *Problem 1*: Formulating and analyzing system specific availability requirements
- *Problem 2*: Lack of integration of the availability specification and analysis into an engineering method

To approach these problems, we introduce an artifact model for availability that allows us to capture system specific availability requirements. We further complement the artifact model with a method to ease the model creation. We validate our approach with a case study in an industrial context.

8.1.1 Availability: State of the Practice

In an interview study with 15 participants from the industry we investigated the current state of the practice regarding availability engineering. To this end we posed four research questions, covering the relevance of availability in the industry, the understanding of availability, the activities related to availability and problems concerning availability. We found the following answers to our research questions:

RQ 1: Relevance – How relevant is the topic availability in the industry? We found that availability of software-intensive systems is a relevant topic in the industry. In several domains (e.g. automation and transportation), availability requirements are specified by customers. Even when not explicitly required, considerable effort is spent to ensure availability. However, the demanded availability always needs to be justified economically.

RQ 2: Understanding – What understandings of availability are present in the industry? We encountered different conceptions of availability, mostly because of different understandings of failure. These differences related to the rigor of failure definitions, the perspective on the system and the classification of failures. We further found that availability is grasped both quantitatively, using availability metrics, or qualitatively, using properties of the system's behavior.

RQ 3: Activities – What availability related activities are performed? The industry performs a large range of availability related activities in different phases of the product's lifecycle. The two areas that receive most attention are system architecture and monitoring. Especially the architecture development is ascribed a great importance for availability. Most of the used strategies to obtain a highly available architecture are based on some form of redundancy. However, other design paradigms, such as "design for failure", also receive attention.

RQ 4: Problems – What problems are perceived regarding availability? The study revealed several problems in the current practice. First, availability is often hard to operationalize and therefore formulating meaningful requirements is difficult. Second, the effort to understand the system and its dependencies, as well as the efforts for applying analysis methods is high. Finally, the modularization of the analysis is difficult and therefore analysis results can hardly be reused.

8.1.2 Availability Artifact Model

To address the problems given in our problem statement, we extend an existing artifact model by models for availability specification and analysis. The extension includes the following four artifacts:

- *Availability Requirements Specification*, containing models for the specification and stepwise formalization of availability requirements
- *Availability Specification*, providing models to specify system specific definitions of failure and calculation rules for availability metrics
- *Extended Logical Architecture*, adapting the original logical architecture to include the system behavior in case of faults
- *Environment Specification*, modeling the structure and behavior of the system environment (e.g. external systems and users)

The included models allow to formulate precise and formal availability requirements, based on system specific notions of failure and availability metrics. They further enable an automated analysis of a system's availability properties.

8.1.3 Availability Modeling Method

To facilitate the practical usage of the artifact method, we complement it with a modeling method. The method includes the following elements.

- *Basic building blocks*, which are parametrizable specifications for availability models. They include specifications for availability metrics, fault injection and behavior comparison.
- A *process*, which describes the sequential creation of availability models, respecting the dependencies between models.
- *Step-by-step guides*, providing guidance for the development of individual models. For example, we suggest a guide word based approach for the elicitation and documentation of failure modes.
- *Modeling patterns*, which provide a basic model structure and thus ease the creation of availability models.

We use an existing example introduced by Broy (2011), a storage and access system, as a running example to illustrate the artifact model and the modeling method.

8.1.4 Case Study: Train Door Control

To evaluate our approach (the artifact model and the modeling method), we apply it to an industrial train control system from Siemens. For the evaluation, we pose three research questions and answer them by performing a case study.

RQ1: Modeling Adequacy – Can the industrial availability requirements be modeled?
The study indicates that both, the artifact model and the modeling method, are applicable and can capture the availability requirements for the train control system. However, it also revealed minor limitations: We could not represent a requirement on the availability in the long run, however, this was a tooling problem and no conceptual issue.

RQ2: Analysis Feasibility – Can the created models be automatically analyzed?
We demonstrate an automated verification of availability requirements, an automated sensitivity analysis to assess the impact of different model parameters, and an automated comparison of architecture alternatives. While the analyses were efficient in terms of memory usage, they took a long time to complete (several hours to days). It is a topic for future research to investigate better analysis techniques for the specific problem.

RQ3: Flexibility – How fragile are our models in the presence of changes in the system?
We found that changes in the functional and logical architecture only resulted in few and small changes in the availability models, indicating that the availability models do not introduce additional redundancy, but add new information instead. At the same time, the analysis results for the changed system differed from the original results, showing that our models reflect the changed availability properties.

8.1.5 Applicability of the Approach

In this thesis, we claim that our approach is applicable to any software-intensive system. Our artifact model for availability builds on and extends a generic artifact model for this kind of systems. We do not presuppose any specific domain, system architecture or description technique. However, we evaluated our approach only for an embedded control system in the rail domain. The specific part of the system that we modelled in the case study focuses more on control and less on data and complex calculations. The failure modes considered by the Siemens engineers could all be expressed using our basic building blocks. For a system that focuses more on data and calculations, for example a complex business information system, creating meaningful failure definitions could be more complex. For such a scenario we might need further, specialized basic building blocks, for example to capture differences in complex data. Nevertheless, the general principles we outlined in this thesis are equally applicable for such a scenario.

A limitation of our approach is that we presuppose a model-based development approach using a specific artifact model as basis. Although model-based development is more and more adopted, in many cases no formal system models are available. While availability requirements can still be formulated using our approach, several parts of step-by-step guides as well as automated analyses cannot be applied in such a setting.

8.2 Outlook

In this section we outline possibilities for future research, improving and extending the contributions of this thesis.

8.2.1 Application to Other Types of Properties

In Section 4.2.3, we described several properties related to availability. Among them are reliability and safety. Although we concentrated on availability in this thesis, the basic idea behind our approach, large parts of the artifact model and parts of the modeling method, can probably be also applied to these properties. Especially for reliability this seems promising, as reliability and availability coincide in case of systems without repair. Probably more work is needed to adapt the approach to safety. Here, the additional concept of a hazard (i.e. a threat to humans or the environment) needs to be integrated. Steps in this direction have been made by Güdemann and Ortmeier (Guedemann and Ortmeier, 2010). However, an integration of these additional properties into a comprehensive engineering method is missing. We provide a first discussion of how to apply our artifact model to reliability and safety in Section 5.7.

8.2.2 Specialized Description Techniques

As description techniques for our models we used the generic techniques also employed to describe systems and system behavior. Examples are data-flow networks, state-transition diagrams or I/O tables. While this works technically, more specific description techniques could be better suited. For example, in order to specify failure definitions, such a description technique might provide abstractions for deviations or failure modes. An example for such a description technique is the Error Annex of the Architecture Analysis and Design Language (AADL) (Society of Automotive Engineers, 2006). This annex provides a language for describing the faults and their effects. It has a similar purpose as the fault-injection models that are part of our artifact model. Developing similar languages for our different availability models would ease the task of creating them and make them more understandable.

8.2.3 Evaluation for Other Domains

In our case study, we evaluate our approach for the railway domain. Although the results indicate that the approach is applicable for this specific domain, further evaluation in other domains is necessary. Especially the domain of business information system would be interesting. For this domain, we expect that other failure modes are relevant and other types of availability metrics are used. Extensions of our method regarding the basic building blocks or the used guide words might be necessary to apply the method in such a setting.

8.2.4 Automated Analysis Techniques

In Chapter 7, we presented a tool prototype for performing availability analyses. The prototype is based on the probabilistic modelchecker Prism. We saw that analyzing the train control system took rather long time. However, there is potential for optimization. One possible direction is to apply advanced probabilistic model checking techniques such as statistical model checking (Brázdil et al., 2014; Hartmanns and Timmer, 2015; Henriques et al., 2012). A different direction is to investigate, whether availability metrics can be iteratively approximated, for example by sequentially considering longer time-intervals until the metric value converges. Performing research in these directions would contribute to the practical applicability of the approach.

Appendix A

Interview Guideline

Question Block	*Question*	
1 General	1.1	What is the task of your department/your project
	1.2	What is your role?
	1.3	What kind of systems are you concerned with or what parts thereof?
2 Relevance of Avail.	2.1	For which kinds of systems and under which circumstances does availability play a role for you?
	2.2	Are there any norms or standards that play a role for availability in your context?
3 Understanding of Avail.	3.1	In your context, what is an outage? When is a system considered as not available?
	3.2	What does "availability" mean in your context? Is there a definition?
	3.3	Do aspects such as degradation or operating modes play a role with respect to availability?
4 Avail. in the PLC	4.1	How is availability taken up in the project management?
	4.2	Do you typically have availability requirements? Which information do availability requirements need to contain?
	4.3	Where do availability requirements origin and how are they documented?
	4.4	How are availability requirements processed during the requirements engineering?

	4.5	How is availability taken up during architecture development? How are availability requirements incorporated?
	4.6	Which possibilities are there to assess the availability through the architecture?
	4.7	How can availability requirements be verified using the architecture?
	4.8	How is availability taken up during implementation? How are availability requirements considered?
	4.9	How can availability requirements be verified?
	4.10	Which role does availability play during maintenance? How are availability requirements considered?
	4.11	Which role does availability play during operation? How are availability requirements considered?
	4.12	Do you perform availability measurements during system operation? If yes, how is availability measured?
	4.13	Across all activities: What means can be employed to describe, ensure and increase availability?
	4.14	Which artifacts are especially important for the description and analysis of availability?
	4.15	Do you use models to describe, realize or analyze availability?
5 Experiences	5.1	Is there a concrete project where availability has played a major role? What were the challenges and problems? What were the root causes for the problems? Which solutions have been found? Have any issues been left open?
	5.2	Which further challenges and problems have emerged in the past? What were the root causes? Which solutions have been found? Have any issues been left open?
6 Closing	6.1	What should a model-based method for the description of availability contain in any case?
	6.2	Which aspects would you add to this interview?
	6.3	Who should we interview additionally?

Bibliography

Allenby, K. and Kelly, T. (2001). Deriving safety requirements using scenarios. In *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering (RE '01)*.

Amazon (2015). Amazon S3 SLA. https://aws.amazon.com/de/s3/sla/. Last Accessed: 2015-02-18.

Anderson, T., Grabbe, T., Hammersley, J., et al. (2001). Providing open architecture high availability solutions. Technical report.

Avižienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1).

Balzer, W. (2015). *Quality of Experience und Quality of Service im Mobilkommunikationsbereich*. Springer.

Basili, V., Donzelli, P., and Asgari, S. (2004). A unified model of dependability: Capturing dependability in context. *IEEE Software*, 21(6).

Bass, L., Clements, P., and Kazman, R. (2013). *Software architecture in practice*. Addison-Wesley.

Bechta Dugan, J., Bavuso, S. J., and Boyd, M. (1992). Dynamic fault-tree models for fault-tolerant computer systems. *IEEE Transactions on Reliability*, 41(3).

Becker, S., Koziolek, H., and Reussner, R. (2009). The Palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1).

Bernardi, S., Flammini, F., Marrone, S., Merseguer, J., Papa, C., and Vittorini, V. (2011a). Model-driven availability evaluation of railway control systems. In Flammini, F., Bologna, S., and Vittorini, V., editors, *Computer Safety, Reliability, and Security*, volume 6894 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg.

Bernardi, S., Merseguer, J., and Lutz, R. (2010). Reliability and availability requirements engineering within the Unified Process using a dependability analysis and modeling profile. In *Proceedings of the 2010 European Dependable Computing Conference (EDCC '10)*.

Bernardi, S., Merseguer, J., and Petriu, D. (2011b). A dependability profile within MARTE. *Software & Systems Modeling*, 10(3).

Bernardi, S., Merseguer, J., and Petriu, D. C. (2012). Dependability modeling and analysis of software systems specified with UML. *ACM Computing Surveys*, 45(1).

Birolini, A. (2010). *Reliability Engineering: Theory and Practice*. Springer Berlin Heidelberg.

Bishop, M. (2012). *Computer security: art and science*. Addison-Wesley.

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)*.

Bolchini, C., Pomante, L., Salice, F., and Sciuto, D. (2001). Reliability properties assessment at system level: a co-design framework. In *Proceedings of the 7th International On-Line Testing Workshop (IOLTW'01)*.

Bondavalli, A. and Simoncini, L. (1990). Failure classification with respect to detection. In *Proceedings of the 2nd IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'90)*.

Botaschanjan, J. and Hummel, B. (2009). Specifying the worst case: orthogonal modeling of hardware errors. In *Proceedings of the eighteenth international symposium on Software testing and analysis (ISSTA '09)*.

Bourque, P. and Fairley, R. E. (2014). *Guide to the Software Engineering Body of Knowledge (SWEBOK): Version 3.0*. IEEE Computer Society Press, Los Alamitos, CA, USA.

Bracquemond, C. and Gaudoin, O. (2003). A survey on discrete lifetime distributions. *International Journal of Reliability, Quality and Safety Engineering*, 10(01).

Breitling, M. (2000). Modeling faults of distributed, reactive systems. In Joseph, M., editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1926 of *Lecture Notes in Computer Science*. Springer.

Breitling, M. (2001). *Formale Fehlermodellierung für verteilte reaktive Systeme*. PhD thesis, Technische Universität München.

Brosch, F. (2012). *Integrated Software Architecture-Based Reliability Prediction for IT Systems*. PhD thesis.

Brosch, F., Koziolek, H., Buhnova, B., and Reussner, R. (2012). Architecture-based reliability prediction with the Palladio component model. *IEEE Transactions on Software Engineering*, 38(6).

Broy, M. (1998). A functional rephrasing of the assumption/commitment specification style. *Formal Methods in System Design*, 13(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. (2011). Seamless method- and model-based software and systems engineering. In Nanz, S., editor, *The Future of Software Engineering*. Springer Berlin Heidelberg.

Broy, M. (2015). Rethinking nonfunctional software requirements. *Computer*, 48(5).

Broy, M. and Stølen, K. (2001). *Specification and development of interactive systems: FOCUS on streams, interfaces, and refinement*. Springer New York.

Brázdil, T., Chatterjee, K., Chmelík, M., Forejt, V., Křetínský, J., Kwiatkowska, M., Parker, D., and Ujma, M. (2014). Verification of markov decision processes using learning algorithms. In Cassez, F. and Raskin, J.-F., editors, *Automated Technology for Verification and Analysis*, volume 8837 of *Lecture Notes in Computer Science*. Springer International Publishing.

Buckl, C. (2008). *Model-Based Development of Fault-Tolerant Real-Time Systems*. Dissertation, Technische Universität München, München.

Chatterjee, K., Doyen, L., and Henzinger, T. (2009). Expressiveness and closure properties for quantitative languages. In *24th IEEE Symposium on Logic In Computer Science (LICS '09)*.

Chung, L., Nixon, B. A., Yu, E., and Mylopoulos, J. (2012). *Non-functional requirements in software engineering*, volume 5. Springer Science & Business Media.

Cysneiros, L. and do Prado Leite, J. (2004). Nonfunctional requirements: from elicitation to conceptual models. *IEEE Transactions on Software Engineering,*, 30(5).

Deissenboeck, F., Wagner, S., Pizka, M., Teuchert, S., and Girard, J. (2007). An activity-based quality model for maintainability. In *Proceedings of the 2007 IEEE International Conference on Software Maintenance (ICSM '07)*.

Despotou, G. and Kelly, T. (2006). Extending safety deviation analysis techniques to elicit flexible dependability requirements. In *Proceedings of the 1st IET International Conference on System Safety*.

Distefano, S. and Puliafito, A. (2009). Reliability and availability analysis of dependent–dynamic systems with DRBDs. *Reliability Engineering & System Safety*, 94(9).

Domis, D. and Trapp, M. (2009). Component-based abstraction in fault tree analysis. In Buth, B., Rabe, G., and Seyfarth, T., editors, *Computer Safety, Reliability, and Security*, volume 5775 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg.

Donzelli, P. and Basili, V. (2006). A practical framework for eliciting and modeling system dependability requirements: Experience from the NASA high dependability computing project. *Journal of Systems and Software*, 79(1).

Dudley, R. (2002). *Real Analysis and Probability*. Cambridge Studies in Advanced Mathematics. Cambridge University Press.

Eusgeld, I., Fraikin, F., Rohr, M., Salfner, F., and Wappler, U. (2008). Software reliability. In Eusgeld, I., Freiling, F., and Reussner, R., editors, *Dependability Metrics*, volume 4909 of *Lecture Notes in Computer Science*, pages 104–125. Springer Berlin Heidelberg.

Fenelon, P., McDermid, J. A., Nicolson, M., and Pumfrey, D. J. (1994). Towards integrated safety analysis and design. *ACM SIGAPP Applied Computing Review*, 2(1).

Ge, X., Paige, R., and McDermid, J. (2009). Probabilistic failure propagation and transformation analysis. In Buth, B., Rabe, G., and Seyfarth, T., editors, *Computer Safety, Reliability, and Security*, volume 5775 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg.

Glinz, M. (2007). On non-functional requirements. In *Proceddings of the 15th IEEE International Requirements Engineering Conference (RE '07)*.

Goševa-Popstojanova, K. and Trivedi, K. S. (2001). Architecture-based approach to reliability assessment of software systems. *Performance Evaluation*, 45(2–3).

Gokhale, S. S. (2007). Architecture-based software reliability analysis: Overview and limitations. *IEEE Transactions on Dependable and Secure Computing*, 4(1).

Goseva-Popstojanova, K., Mathur, A., and Trivedi, K. (2001). Comparison of architecture-based software reliability models. In *Proceedings of the 12th International Symposium on Software Reliability Engineering (ISSRE '01)*.

Gray, J. and Siewiorek, D. (1991). High-availability computer systems. *Computer*, 24(9).

Grottke, M., Sun, H., Fricks, R., and Trivedi, K. (2008). Ten fallacies of availability and reliability analysis. In Nanya, T., Maruyama, F., Pataricza, A., and Malek, M., editors, *Service Availability*, volume 5017 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg.

Grunske, L. (2006). Towards an integration of standard component-based safety evaluation techniques with saveccm. In Hofmeister, C., Crnkovic, I., and Reussner, R., editors, *Quality of Software Architectures*, volume 4214 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg.

Grunske, L., Lindsay, P., Bondarev, E., Papadopoulos, Y., and Parker, D. (2007). An outline of an architecture-based method for optimizing dependability attributes of software-intensive systems. In de Lemos, R., Gacek, C., and Romanovsky, A., editors, *Architecting Dependable Systems IV*, volume 4615 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg.

Grunske, L. and Zhang, P. (2009). Monitoring probabilistic properties. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering (ESEC/FSE '09)*. ACM.

Guedemann, M. and Ortmeier, F. (2010). Probabilistic model-based safety analysis. In *Proceedings of the 8thWorkshop on Quantitative Aspects of Programming Language (QAPL '10)*.

Hartmanns, A. and Timmer, M. (2015). Sound statistical model checking for MDP using partial order and confluence reduction. *International Journal on Software Tools for Technology Transfer*, 17(4).

Henriques, D., Martins, J., Zuliani, P., Platzer, A., and Clarke, E. (2012). Statistical model checking for markov decision processes. In *Proceedings of the 9th International Conference on Quantitative Evaluation of Systems (QEST '12)*.

Henzinger, T. A. (2010). From boolean to quantitative notions of correctness. In *Proceedings of the 37th Symposium on Principles of Programming Languages (POPL '10)*. ACM.

IEC (1990). IEC 61025 Fault Tree Analysis.

IEC (1996). IEC 60300-3-4 Dependability management: Part 3-4: Application guide—Guide to the specification of dependability requirements.

IEC (2004). IEC 60300-3-3 Dependability management - Part 3-3: Application guide - Life cycle costing.

IEC (2006). IEC 62347:2006 Guidance on system dependability specifications.

IEC (2012). IEC 62741 Reliability of systems, equipment and components. Guide to the demonstration of dependability requirements. The dependability case.

IEEE (2007). IEEE Std 1471-2000 Systems and software engineering - recommended practice for architectural description of Software-Intensive Systems.

Immonen, A. and Niemelä, E. (2008). Survey of reliability and availability prediction methods from the viewpoint of software architecture. *Software and Systems Modeling*, 7(1):49–65.

ISO (2011). ISO 26262 Road vehicles – Functional safety – Part 1: Vocabulary.

ISO/IEC (2011). ISO/IEC 25010:2011 Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software qualityISO/IEC.

ITU (2002). G.826 (12/02): End-to-end error performance parameters and objectives for international, constant bit-rate digital paths and connections.

ITU (2008). ITU-T E.800 E.800 : Definitions of terms related to quality of service.

Janakiraman, G. J., Santos, J. R., and Turner, Y. (2004). Automated system design for availability. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN '04)*. IEEE Computer Society.

Janicak, C. (2009). *Safety Metrics: Tools and Techniques for Measuring Safety Performance*. Government Institutes.

Jhumka, A., Klaus, S., and Huss, S. (2005). A dependability-driven system-level design approach for embedded systems. In *Proceedings of 2005 Design, Automation and Test in Europe (DATE '05)*.

Johannessen, P., Grante, C., Alminger, A., Eklund, U., and Torin, J. (2001). Hazard analysis in object oriented design of dependable systems. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks (DSN '01)*.

Johnson, R. B. (1997). Examining the validity structure of qualitative research. *Education*, 118(2).

Joshi, A., Vestal, S., and Binns, P. (2007). Automatic generation of static fault trees from AADL models. In *Proceedings of the 2007 Workshop on Architecting Dependable Systems (WADL '07)*.

Junker, M. (2014). Exploiting behavior models for availability analysis of interactive systems. In *Proceedings of the 2014 IEEE International Symposium on Software Reliability Engineering (ISSRE '14)*.

Junker, M. and Neubeck, P. (2012). A rigorous approach to availability modeling. In *Proceedings of the 2012 ICSE Workshop on Modeling in Software Engineering (MiSE '12)*.

Kaiser, B., Liggesmeyer, P., and Mäckel, O. (2003). A new component concept for fault trees. In *Proceedings of the 8th Australian Workshop on Safety Critical Systems and Software (SCS '03)*.

Kaiya, H., Horai, H., and Saeki, M. (2002). AGORA: attributed goal-oriented requirements analysis method. In *Proceedings of the 10th International Conference on Requirements Engineering (RE '02)*.

Kan, S. H. (2002). *Metrics and models in software quality engineering*. Addison-Wesley Longman Publishing.

Kaniche, M., Kanoun, K., and Martinello, M. (2003). A user-perceived availability evaluation of a web based travel agency. In *Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN '03)*.

Kitchenham, B., Linkman, S., Pasquini, A., and Nanni, V. (1997). The SQUID approach to defining a quality model. *Software Quality Journal*, 6(3).

Kitchenham, B. A. and Pfleeger, S. L. (2008). Personal opinion surveys. In Shull, F., Singer, J., and Sjøberg, D. I., editors, *Guide to Advanced Empirical Software Engineering*. Springer London.

Kletz, T. (1999). *HAZOP and HAZAN: Identifying and assessing process industry hazards*. Institution of Chemical Engineers.

Kubat, P. (1989). Assessing reliability of modular software. *Operations Research Letters*, 8(1).

Kwiatkowska, M., Norman, G., and Parker, D. (2011). PRISM 4.0: Verification of probabilistic real-time systems. In Gopalakrishnan, G. and Qadeer, S., editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*. Springer.

Lapp, S. A. and Powers, G. J. (1977). Computer-aided synthesis of fault-trees. *IEEE Transactions on Reliability*, R-26(1).

Laprie, J.-C. (1984). Dependability evaluation of software systems in operation. *IEEE Transactions on Software Engineering*, SE-10(6).

Laprie, J.-C. (1995). Dependable computing and fault tolerance : Concepts and terminology. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing (FTCS '95)*.

Littlewood, B. (1979). Software reliability model for modular program structure. *IEEE Transactions on Reliability*, R-28(3).

Lochmann, K. (2014). *Defining and Assessing Software Quality by Quality Models*. Dissertation, Technische Universität München.

Majdara, A. and Wakabayashi, T. (2009). Component-based modeling of systems for automated fault tree generation. *Reliability Engineering & System Safety*, 94(6).

McDermid, J. and Pumfrey, D. (1994). A development of hazard analysis to aid software design. In *Proceedings of the 9th Annual Conference on Computer Assurance (COMPASS '94)*.

Milanovic, N. and Milic, B. (2011). Automatic generation of service availability models. *IEEE Transactions on Services Computing*, 4(1).

Musa, J. (1993). Operational profiles in software-reliability engineering. *Software*, 10(2).

Musa, J. D. (1996). The operational profile. In Özekici, S., editor, *Reliability and Maintenance of Complex Systems*, volume 154 of *NATO ASI Series*. Springer Berlin Heidelberg.

Neubeck, P. (2012). *A Probabilitistic Theory of Interactive Systems*. Dissertation, Technische Universität München.

Newman, R. C. (2009). *Computer security: Protecting digital resources*. Jones & Bartlett Publishers.

OMG (2009). UML profile for MARTE: Modeling and analysis of real-time embedded systems.

Papadopoulos, Y. and Maruhn, M. (2001). Model-based synthesis of fault trees from matlab-simulink models. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks (DSN '01)*.

Pfleeger, C. P. and Pfleeger, S. L. (2006). *Security in Computing (4th Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA.

Pohl, K., Hönninger, H., Achatz, R., and Broy, M. (2012). *Model-based Engineering of Embedded Systems: The SPES 2020 Methodology*. Springer.

Powell, D. (1995). Failure mode assumptions and assumption coverage. In Randell, B., Laprie, J.-C., Kopetz, H., and Littlewood, B., editors, *Predictably Dependable Computing Systems*. Springer Berlin Heidelberg.

Pumfrey, D. J. (1999). *The principled design of computer system safety analyses*. PhD thesis, University of York.

Rausand, M. and Høyland, A. (2004). *System Reliability Theory: Models, Statistical Methods and Applications*. Wiley-Interscience, Hoboken, NJ.

Reussner, R. H., Schmidt, H. W., and Poernomo, I. H. (2003). Reliability prediction for component-based software architectures. *Journal of Systems and Software*, 66(3).

Rossebeø, J. E., Lund, M. S., Husa, K. E., and Refsdal, A. (2006). A conceptual model for service availability. In Gollmann, D., Massacci, F., and Yautsiukhin, A., editors, *Quality of Protection*, volume 23 of *Advances in Information Security*. Springer US.

Runeson, P. and Regnell, B. (1998). Derivation of an integrated operational profile and use case model. In *Proceedings of the Ninth International Symposium on Software Reliability Engineering (ISSRE '98)*.

Saridakis, T. (2002). A system of patterns for fault tolerance. In *Proceedings of 7th European Conference on Pattern Languages of Programs (EuroPloP '02)*.

Schilling, R. (2005). *Measures, Integrals and Martingales*. Cambridge University Press.

Scott, J. and Kazman, R. (2009). Realizing and refining architectural tactics: Availability. Technical Report CMU/SEI-2009-TR-006.

Shukla, R., Carrington, D., and Strooper, P. (2004). Systematic operational profile development for software components. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC '04)*.

Society of Automotive Engineers (2006). SAE AS 5506/1 – SAE architecture analysis and design language (AADL) annex volume 1.

Society of Automotive Engineers (2012). SAE AS 5506B – architecture analysis & design language (AADL).

Sommerville, I. and Sawyer, P. (1997). *Requirements engineering: a good practice guide*. John Wiley & Sons, Inc.

Streichert, T., Glaß, M., Haubelt, C., and Teich, J. (2007). Design space exploration of reliable networked embedded systems. *Journal of Systems Architecture*, 53(10).

Tokuno, K. and Yamada, S. (2008). User-perceived software service availability modeling with reliability growth. In Nanya, T., Maruyama, F., Pataricza, A., and Malek, M., editors, *Service Availability*, volume 5017 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg.

Trivedi, K., Wang, D., and Hunt, J. (2010). Computing the number of calls dropped due to failures. In *Proceedings of the 21st International Symposium on Software Reliability Engineering (ISSRE '10)*.

Van Lamsweerde, A. (2001). Goal-oriented requirements engineering: A guided tour. In *Proceedings of the fifth IEEE International Symposium on Requirements Engineering (RE '01)*.

Vogelsang, A. (2015). *Model-based Requirements Engineering for Multifunctional Systems*. Dissertation, Technische Universität München.

Wang, D. and Trivedi, K. (2005). Modeling user-perceived service availability. In Malek, M., Nett, E., and Suri, N., editors, *Service Availability*, volume 3694 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg.

Withall, S. (2007). *Software Requirement Patterns*. Pearson Education.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2012). *Experimentation in software engineering*. Springer.

Xie, Y., Li, L., Kandemir, M., Vijaykrishnan, N., and Irwin, M. (2004). Reliability-aware co-synthesis for embedded systems. In *Proceedings of the 15th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP '04)*.

Yacoub, S., Cukic, B., and Ammar, H. (2004). A scenario-based reliability analysis approach for component-based software. *IEEE Transactions on Reliability*, 53(4).