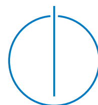
Software Deployment Analysis for Mixed Reliability Automotive Systems

Klaus Becker

fortiss



Technical University of Munich

Institut für Informatik
der Technischen Universität München

Software Deployment Analysis for Mixed Reliability Automotive Systems

Klaus Becker

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: Prof. Dr.-Ing. Georg Carle

Prüfer der Dissertation:

1. Prof. Dr. Dr. h.c. Manfred Broy
2. Prof. Dr.-Ing. Jürgen Teich, Universität Erlangen-Nürnberg

Die Dissertation wurde am 16.02.2017 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 02.06.2017 angenommen.

Abstract

Safety critical fault-tolerant embedded systems have to react properly on failures of internal system elements to avoid failure propagation and finally an external failure at the entire system boundary, being harmful to other systems, material or human beings in the environment. To inhibit failure propagation and ensure harm avoidance, failing system elements have to be isolated from the remaining system. As isolated elements become unavailable, the amount of available resources, like execution units or input data, decreases. This may cause that the system resources become insufficient to provide the entire initial set of functional features. However, it is often not allowed that the system stops its operation completely in such situations. Instead, the set of provided functional features of the system should be degraded gracefully to keep available at least a subset of the intended features. In this context, a system shall keep available those features as long as possible, that have the highest requirements according to safety, reliability and availability. For instance, features having a fail-operational requirement must be kept available, while other features may be allowed to get deactivated, resulting in a degradation of the available set of features.

In this thesis, we introduce an approach to formally analyze whether the design of a system adheres to all fail-operational requirements of functional features. We consider this in different scenarios of failing system elements, like failing execution units and failing software components. We address mixed criticality and mixed reliability automotive systems, comprising functional features having varying requirements to be fail-operational. Beside pure fail-operational features, we also consider degradable functional features, called fail-degraded features. As prerequisite for the analysis, we set up a formal model of the system, containing the functional features, possible feature degradations, the relationship of functional features to the software components that realize the features, the communication dependencies between software components, the hardware execution units, as well as the deployment of software components to execution units. For assumed failure scenarios, we provide a structural analysis of the necessary level of degradation in these scenarios by analyzing which subset of functional features can be kept available. As part of our analysis, we automatically synthesize valid deployments of software components to execution units, fulfilling the fail-operational requirements of all functional features in all failure scenarios, and minimizing the required level of degradation. We incorporate an adequate level of redundancy into the deployment, to enable failover mechanisms from an isolated software component to a redundant backup of that component, if this is necessary due to a fail-operational requirement. We set up formal constraints which describe valid redundant deployments of software components to execution units, as well as valid degradation scenarios and valid failover scenarios by describing valid reconfigurations of deployments. This means, we determine an initial deployment and analyze the changes of this deployment that may become required in different failure scenarios. The formal model, constraints and optimization objectives are described by linear arithmetic and logical operators. We use an SMT solver to get solutions for this model, serving as basis for our analysis. We show the applicability of our approach based on three examples from the automotive domain.

Acknowledgements

First of all, I want to express my deep gratitude for Prof. Manfred Broy for his outstanding supervision of this thesis, supporting me with so many discussions and constructive review comments. It was always a pleasure to experience his style of open-minded research and guidance. I also thank Prof. Jürgen Teich for co-supervising my thesis.

I express a particular thank you to Michael Armbruster of Siemens AG, for inspiring many of the requirements that are analyzed in this thesis, for raising my interest in redundancy and fault-tolerance concepts, and responding fruitful feedback to the very early ideas about this work. Great thanks also goes to Bernhard Schätz and Sebastian Voss of fortiss Institute, supporting me in a lot of discussions and providing feedback to contents of this thesis.

I also would like to thank all the colleagues who worked together with me in the RACE project, for the very intensively cooperative work during the project and also discussing early ideas about this work, particularly again Michael Armbruster, Christian Buckl, Jelena Frtunikj, Cornel Klein, Ludger Fiege and Meik Felser, who were with Siemens AG or fortiss Institute during that time.

I also would like to thank my other colleagues of TU Munich and fortiss Institute, in particular Andreas Vogelsang, Maximilian Junker, Mario Gleirscher, Stefan Kugele, Denis Bytschkow and Vincent Aravantinos, for their interest, discussions and feedback around the work in this thesis.

A thank you also goes to Nikolaj Bjørner of Microsoft Research for his support by quickly fixing some issues in the Z3 SMT solver, revealed during the implementation of the introduced model and analysis approach.

The time as PhD candidate is not only about writing the thesis, but furthermore obtaining experiences in several other research topics and projects, which turn the time into a very enjoyable and beneficial time! I would like to thank Ernst Denert, Gero Scholz, Alois Zoitl and all the other people with which I worked together during my time as researcher at fortiss Institute, for sharing a lot of experiences and knowledge in many different areas.

Last but not least, the most important people who supported me so much during this time and missed me so often or tried to do not disturb me while writing, my wife Sandra and - at the end - our little cheerful son Jonas, I love you two.

Contents

1 Introduction **1**
- 1.1 Context . . . 1
- 1.2 Problem Statement and Motivation . . . 1
- 1.3 Research Questions and Contribution . . . 3
- 1.4 Outline . . . 5

2 Fundamentals **7**
- 2.1 Terms and Definitions . . . 7
  - 2.1.1 Dependability . . . 9
  - 2.1.2 Dependability Threats . . . 10
  - 2.1.3 Dependability Attributes . . . 12
  - 2.1.4 Dependability Means . . . 13
  - 2.1.5 Other Definitions related to Dependability . . . 15
  - 2.1.6 Redundancy and Replication Mechanisms . . . 18
- 2.2 Foundations in Safety Engineering . . . 23
- 2.3 Foundations in Software and System Quality Assurance . . . 24
- 2.4 Automotive Architectures and Standards . . . 25
  - 2.4.1 Classical Automotive E/E Architectures . . . 25
  - 2.4.2 AUTOSAR . . . 25
  - 2.4.3 ISO 26262 . . . 29
- 2.5 The RACE Approach . . . 30
  - 2.5.1 Software and System Architecture . . . 30
  - 2.5.2 Safety and Fault-Tolerance Concept . . . 33
  - 2.5.3 Application Development and RTE Configuration . . . 36
  - 2.5.4 Demonstrator Vehicles . . . 37
- 2.6 Avionic Architectures and Standards . . . 37
- 2.7 Foundations in Satisfiability Solving and Optimization . . . 38

3 Related Work **41**
- 3.1 Approaches to Design and Analyze Fault-Tolerant Systems . . . 41
  - 3.1.1 Design and Analysis of Graceful Degradation . . . 42
  - 3.1.2 Design and Analysis of Reliability and Robustness . . . 48
  - 3.1.3 Design and Analysis of Availability . . . 51
  - 3.1.4 Fault-Tolerant Scheduling for Mixed Criticality Systems . . . 51
  - 3.1.5 Design of Structural and Behavioral Reconfiguration . . . 52
  - 3.1.6 Self-x Approaches . . . 53
- 3.2 Constraint Based Synthesis of Design Decisions . . . 58

4 Analyzing Fail-Operational and Fail-Degrading Systems **65**
- 4.1 Introduction to the Formal System Model . . . 66
  - 4.1.1 Viewpoints . . . 66
  - 4.1.2 Meta-Model of Considered System Structure . . . 67
  - 4.1.3 Motivation and Benefits of the Formal System Model . . . 69

4.2 Formal System Model . . . 70
4.2.1 Functional Features . . . 72
4.2.2 Software Architecture . . . 73
4.2.3 Feature Realization . . . 78
4.2.4 Hardware Architecture . . . 78
4.2.5 System Configuration . . . 79
4.2.6 System Model . . . 80
4.2.7 Example for the Formal Definitions . . . 80
4.2.8 Summary Overview of Formal Model Symbols . . . 82
4.3 Concept Overview . . . 83
4.4 Properties of System Model Elements . . . 84
4.4.1 Overview of Properties of Formal System Model . . . 85
4.4.2 Input Model Properties . . . 86
4.4.3 Solution Model Properties for Initial Deployment . . . 88
4.4.4 Solution Model Properties for Failure Scenarios . . . 92
4.4.5 States of ASWC Instances . . . 93
4.5 Synthesis of Valid Redundant Deployments . . . 95
4.5.1 Formal Constraints for Valid Redundant Deployments . . . 95
4.5.2 Examples . . . 98
4.6 Analysis of Failure Effects . . . 98
4.6.1 Scenarios . . . 99
4.6.2 The Scenario Graph . . . 99
4.6.3 Extensions of Model Properties to Cover the Scenarios . . . 101
4.6.4 Procedure to Analyze the Scenario Graph . . . 102
4.6.5 Formal Constraints for Valid Failovers and Degradations . . . 103
4.6.6 Formal Constraints to Analyze Feature Availability . . . 105
4.6.7 Relaxation of Constraints to Localize Problems . . . 106
4.6.8 Example A – Basic Example . . . 108
4.6.9 Example B – Communication Channels . . . 113
4.6.10 Size of the Scenario Graph . . . 119
4.7 Supporting Degradations of Single Functional Features . . . 122
4.7.1 Assumed Design Principles for Feature Degradations . . . 122
4.7.2 Extension of the Formal System Model . . . 123
4.7.3 Extended Overview of Properties . . . 127
4.7.4 Formal Constraints for Feature Degradations . . . 128
4.7.5 Example C – Feature Degradation . . . 129
4.8 Formalization of Optimization Objectives . . . 140
4.9 Assumptions and Aspects that are out of Scope . . . 142
4.9.1 Out of scope . . . 142
4.9.2 Assumed properties of the system under analysis . . . 142
4.10 Explanations and Discussions about the Formal System Model . . . 144
4.10.1 Functional Features . . . 144
4.10.2 Software Architecture . . . 145
4.10.3 Feature Realization . . . 150
4.10.4 Hardware Architecture . . . 150
4.10.5 System Configuration . . . 150

5 Evaluation **153**
5.1 Discussion of Research Questions 153
5.2 Limitations of the presented approach 155
5.3 Threats to Validity 157

6 Conclusions and Future Work **159**
6.1 Summary and Conclusions 159
6.2 Out of Scope 161
6.3 Future Work 161
6.3.1 Expand the set of considered design aspects 162
6.3.2 Possible Future Work Extensions of our Approach 162
6.3.3 Evaluation of alternative solving and optimization strategies 163

A Appendix **165**
A.1 Input Files of Examples 165

List of Figures **175**

List of Tables **177**

Bibliography **178**

CHAPTER 1 Introduction

Contents

1.1 Context . . . 1
1.2 Problem Statement and Motivation . . . 1
1.3 Research Questions and Contribution . . . 3
1.4 Outline . . . 5

1.1 Context

Complex systems, like automobiles, airplanes or industrial automation systems, are nowadays realized by a huge amount of software, rather than using purely mechanical, electrical or hydraulical components like half a century ago. We call such systems to be software-intensive embedded systems, in which software based subsystems interact with the physical world using sensors and actuators to fulfill their intended service. An example is an anti-lock braking system (ABS) of a vehicle, which senses the wheel rotations and software decides about how to act on the brakes to keep wheels rotating and avoid slipping wheels. More and more of such software-based functional features become integrated into vehicles, like advanced driver assistance systems (ADAS) and automated/autonomous driving capabilities [115]. This means that the amount of software in such systems is growing and increasingly interconnected, leading to increasing complexity [216].

Many embedded systems operate in safety-critical environments, like in the automotive domain, avionics domain or industrial automation domain. Those systems have critical properties in a sense that unhandled malfunctions of system parts may lead to unacceptable harms to the system itself, to the physical system environment and even to the safety of human beings. Hence, it is very important that those malfunctions are detected and handled in a safe manner to avoid any form of harm. This thesis provides an approach to analyze such safety critical systems with respect to the set of functional features, that a system can still provide when it becomes affected by malfunctions.

1.2 Problem Statement and Motivation

Safety critical systems need to contain mechanisms to detect malfunctions of system parts and to react on these malfunctions properly such that no harm can occur with impact to material or life. In case software or hardware elements of a system fail, these elements have to be isolated from the residual system in order to avoid a failure propagation through the residual system, which potentially leads to a total system loss and harm.

In order to develop vehicles with automated or fully autonomous driving capabilities, such systems are becoming highly integrated and interconnected. The functionalities get more and more based on X-by-Wire control using electronics and software, rather than mechanical or hydraulic control components [47]. Examples are Fly-by-Wire in aircraft (e. g., in [360]) or Drive-by-Wire in vehicles (e. g., in [46]). As a disappearance of active assistance or autonomous driving features during their operation would be very dangerous causing harm, these features must be kept alive even in the presence of a malfunction of some system element [56] [331] [143]. Due to these safety reasons, fully autonomous X-by-Wire systems

must behave fail-operational, as fail-safe behavior would cause the loss of autonomy features, which may be harmful if the driver is not able to take over the control fast enough. Hence, X-by-Wire systems require a fail-operational implementation, fail-safe is not sufficient [203]. To ensure fail-operationality, such systems have to be established with redundant backup elements that can take over functionality, if another element fails [189]. Redundancy helps to establish error avoidance [276] and by this, to provide fail-operational systems. If a X-by-Wire system is desired to have no mechanical fallback backups of software enabled features, software based backups by redundancy mechanisms are important to ensure fail-operationality of that features. Beside software redundancy, also redundancy of execution hardware as well as sensors and actuators is required to handle electronic hardware failures [322]. But any form of redundancy is limited and costly. Hence, graceful degradation and fail-safe mechanisms have also to be taken into account simultaneously.

Furthermore, a trend is towards mixed-criticality systems [73], meaning that components with different levels of criticality are executed by the same hardware device in modern systems. It has to be guaranteed that errors of low critical components can never have negative impact on highly critical components.

But what should happen after a failure of a system element has been detected? Invalidating faulty data and going into a fail-safe state may cause the loss of functional features. This is not acceptable for critical features that require fail-operational behavior, meaning that those features must be kept alive even in the presence of failures of system elements, like hardware or software components. If a fail-operational feature gets affected by such a failure, the systems must be able to resume the feature without any service interruption. For instance, if a hardware failure of an execution unit has been detected, followed by an isolation of this execution unit from the remaining system, another execution unit has to be able to provide at least a subset of those features that were provided by the failed execution unit. However, if system resources get lost due to isolations, the remaining system resources may become insufficient to provide the full set of functional features. Hence, the remaining resources should be used efficiently to keep available the most important features. The importance often corresponds to requirements about safety, reliability and availability. In these scenarios, keeping available some features requires to explicitly deactivate some other features, resulting in a *degradation* of the system.

The motivation of this thesis is to provide an approach to formally analyze a system with respect to needed degradations in failure scenarios, while ensuring the fulfillment of fail-operational requirements. In order to perform such an analysis, an initial deployment of software components to hardware execution units has to be determined, which fulfills a set of redundancy constraints. We aim in supporting the designer by automatically synthesizing such a valid initial deployment, as well as valid reconfigurations of the deployment for all failure scenarios. We want to analyze if deployments can be synthesized that enable the fulfillment of all fail-operational requirements of features by the current system design. We also want to analyze which subset of functional features has to be deactivated in certain failure scenarios, and which subset of featues can be kept available, even if these features do not have fail-operational requirements. As the failure combinations and the reactions to failures can become quite complex, a manual analysis without technical support would be very time intensive, error-prone and inappropriate. An automated analysis during design time would give the system designer an early feedback about problems, helping to reduce costs by early design corrections, and providing evidence towards the fulfillment of different fail-operational requirements of the functional features of the mixed-reliability system.

We assume a given system safety concept that specifies redundancy mechanisms to be applied in the deployment, and specifies isolation and *failover* mechanisms to be used as reaction to detected failures. We aim at providing an automated analysis of the degradation scenarios, resulting from synthesizing a deployment with a minimal level of redundancy needed to ensure all fail-operational requirements.

Apart from analyzing the reaction to failing hardware or software components, another use-case of our approach is to analyze intended degradation scenarios, which are applied in order to save energy in a vehicular network. Such a planned deactivation of resources due to energy saving reasons leads as well to decreasing amount of available resources. Hence, degradations of system functionality should be taken into

account. For instance, AUTOSAR [21] mentions the requirement that *AUTOSAR shall support different standardized methods to degrade the functionality of an AUTOSAR system*.[1] This is mainly motivated to save energy as energy efficiency is a major concern in vehicles [250]. Two main degradation strategies are specified in AUTOSAR: 1) *Partial Networking* and 2) *ECU degradation* in conjunction with *Pretended Networking* [225].

Although fault-detection, fault-handling, and energy-saving degradation have to be done at runtime of the system, it is desireable to be able to statically analyze at design time the current system design according to degradation scenarios that may happen at runtime. Particularly the mixed criticality and mixed reliability of the system requires to ensure runtime robustness and to provide static verification at design time for certification [29]. Providing such a static analysis approach to support the verification with focus on degradation and failover scenarios is the major motivation of this thesis.

1.3 Research Questions and Contribution

In this thesis, we tackle the following research questions and present respective contributions to address the questions.

RQ1: How to automatically calculate valid deployments of software to hardware, supporting the fulfillment of fail-operational requirements? The first research question of this thesis is how valid deployments of mixed critical software components onto hardware execution units can be automatically calculated, incorporating a given redundancy concept and applying an adequate level of redundancy. The redundancy is required to fulfill fail-operational requirements of functional features. Different levels of fail-operationality may be required. Hence, different levels of redundancy of software components are adequate. This results in a fault-tolerant system, whose functional features have *mixed reliability*. The question is how this can be modeled adequately and how appropriate deployments can be synthesized automatically?
Our approach and focus: We do not estimate predefined deployments, but instead we synthesize valid redundant deployments. In order to calculate a valid deployment, we setup a formal system model with formal deployment constraints, expressed with linear arithmetic and logical formulas, enabling an automated calculation of valid deployments that fulfill the constraints. Beside calculating deployments, we also calculate communication channels between the software components, based on component port specifications, opening a set of possible channels from which only a subset of channels is necessary to satisfy the specified subscription ports. This enlarges and influences the design space for the deployments.
Novelty of contribution: Calculating deployments by setting up a *Constraint Satisfaction Problem* (CSP) is itself nothing new, but so far not addressed in other work for the introduced kind of systems with mixed critical and mixed reliable functional features, having different required levels of fail-operationality, allowing to efficiently apply different levels of redundancy of software components, considering constraints for valid types of redundancy. In addition, we use the open design space of deployment and channel selection to optimize the synthesized architecture, for instance by preferring local communication channels during the deployment synthesis, reducing the network traffic between the execution units.

RQ2: How to formally analyze the ability to keep functional features available in scenarios of failing system elements, and decide about necessary degradations of the available feature set, incorporating necessary failovers to ensure fail-operational requirements? Our major research question is how a given system design, including its feature set and software architecture, can be formally analyzed with respect to the degradation of the set of functional features that can be kept available in scenarios of failing

[1] See specification document *"AUTOSAR RS Features"* in *"Auxiliary Material"* at http://www.autosar.org/specifications/release-41/main

system elements, and how functional features with fail-operational requirements can be kept available by applying *failover* mechanisms in such failure scenarios, using redundant backups of software components? This analysis shall be done in combination with the first research question, such that an initial deployment is synthesized that leads to applicable failover scenarios and a minimum level of *degradation* of available features. One sub research question is how to decide which software components are allowed to be explicitly deactivated, in case the available system resources become insufficient to execute all software components?
Our approach and focus: We consider failures of hardware execution units and failures of software components. To be able to analyze the effect of failing hardware or software to the set of available functional features, we formally describe the relationship between functional features and the software components, which realize these features. To decide about explicit deactivations, we establish a *quantitative metric* to estimate the intrinsic value of software components in order to decide about the sequence in which components (and thereby features) should be disabled when the system is not able to provide the full set of functional features anymore. We focus on an analysis on structural architecture level, without doing a behavior analysis.
Novelty of contribution: No existing approach considered a similar form of mixed criticality and mixed reliability fault-tolerant systems. There are existing approaches, providing quantitative metrics about the degree of *graceful degradation* of fault-tolerant systems (e. g., [314]), but the approach shown in this thesis has a novel combination of mixed criticality and mixed reliability (by different fail-operational requirements of functional features) and a system concept using global redundancy and failover mechanisms.

RQ3: How to formalize a given system design concept and the requirements related to the safety and fault-tolerance concept of that system to be able to apply the formal analysis to this type of system? In order to tackle the first two research questions, one prerequisite is the design of a formal system model that represents the hardware and software architecture of the system, as well as the relevant properties of the system elements. Applying a given safety and fault-tolerance concept with informal natural language descriptions of valid types of redundancy, valid failover mechanisms and valid degradation scenarios, the question is how to formally express these descriptions in form of formal constraints over the formal system model? The aim is to synthesize valid respectively optimal solutions in a mathematical fashion. This means, informal requirements have to be transformed into formal constraints over the elaborated formal model, expressing valid types of architectures and reconfigurations of it, to be applied in scenarios of failing system elements. To formally synthesize valid deployments and analyze failure scenarios for a *system under analysis*, this requires to formalize parts of the applied safety and fault-tolerance concept. The research question is how this can be done.
Our approach and focus: As example safety and fault-tolerance concept for embedded systems, specifying informal requirements for redundancy and failover mechanisms, we use the concept that was developed in the research project RACE. [2] For an introduction into the RACE platform concepts, see section 2.5.1. We formalize relevant aspects of the safety and fault-tolerance concept by formal constraints over the introduced system model. The constraints ensure that the solutions, calculated by an employed solver, represent failover and degradation scenarios that are valid according to the given informal requirements.
Novelty of contribution: No existing work has been found that provides a combined formalization of architecture concepts and requirements for valid redundancy, failover and degradation mechanisms to enable fault-tolerance in the above described manner.

[2] The Project RACE (Robust and Reliant Automotive Computing Environment for Future eCars) was funded by the German Federal Ministry for Economic Affairs and Energy (BMWi) under grant no. 01ME12009 from 2012 to 2015, `http://www.projekt-race.de/en`

Further motivation for the presented approach: Beside the three major research questions, motivating the approach introduced in this thesis, there are some sub-motivations that influence the introduced approach:

- *Static analysis at design time:* We tackle how dynamic failover scenarios that may happen at runtime can be encoded into a formal model to be able to analyze these scenarios statically at *design time*. Our work contributes to provide such an analysis, applicable during design time of a system. It is not in focus to apply our approach during runtime of the system executed by the system itself, as this changes the requirements with respect to the performance of calculating solutions and the workflow.
- *Optimization objectives:* Another question is how optimization objectives can be expressed based on the formal system model, and be solved reusing existing technologies like for instance *Satisfiability Modulo Theories* (SMT) or *Mixed Integer Linear Programming* (MILP) solvers? If necessary, also partially contradicting optimization objectives should be expressible and solvable.

In this thesis, we express the tackled problem as a formal system model, using arithmetic and logical formulas, and use the model as input for an SMT solver, which calculates valid values of solution variables satisfying formalized constraints, serving as basis for our analysis, while optimizing some considered design objectives. We use an SMT solver with optimization facilities to do this.

1.4 Outline

The residual part of this thesis is structured as follows. Chapter 2 introduces the fundamental terminology which is present in the context of this thesis, as well as an overview over foundations in safety engineering and related architectures and standards of the automotive domain. Chapter 3 discusses the related work of this thesis. Chapter 4 contains the presentation of the contribution of this thesis, including our formal system model and the analysis approach for failure scenarios, which is evaluated in chapter 5. The conclusion and future work is shown in chapter 6. Additional material is contained in appendix chapter A.

Previously Published Material

Parts of the contributions presented in this thesis have been published in the following papers directly related to this thesis: [39], [35], [38], [37], [40], [41].

[39] Klaus Becker and Sebastian Voss. Towards Dynamic Deployment Calculation for Extensible Systems using SMT-Solvers. In First Open EIT ICT Labs Workshop on Cyber-Physical Systems Engineering (EIT CPSE), Trento, Italy, May 2013.

[35] Klaus Becker, Michael Armbruster, Bernhard Schätz, and Christian Buckl. Deployment calculation and analysis for a fail-operational automotive platform. In 1st Workshop on Engineering Dependable Systems of Systems (EDSoS). arXiv:1404.7763, 2014.

[38] Klaus Becker, Bernhard Schätz, Michael Armbruster, and Christian Buckl. A formal model for constraint-based deployment calculation and analysis for fault-tolerant systems. In 12th Int. Conference on Software Engineering and Formal Methods (SEFM), volume 8702, pages 205–219. Springer Lecture Notes in Computer Science (LNCS), 2014.

[37] Klaus Becker and Bernhard Schätz. Deployment calculation and analysis for a fault-tolerant system platform. In 11th Dagstuhl-Workshop on Model-Based Development of Embedded Systems (MBEES), 2015.

[40] Klaus Becker and Sebastian Voss. Analyzing graceful degradation for mixed critical fault-tolerant real-time systems. In 18th International Symposium on Real-Time Distributed Computing (ISORC), pages 110–118. IEEE, 2015.

[41] Klaus Becker and Sebastian Voss. A formal model and analysis of feature degradation in fault-tolerant systems. In 4th Int. Workshop on Formal Techniques for Safety-Critical Systems (FTSCS), volume 596, pages 139–154. Springer Communications in Computer and Information Science (CCIS), 2015.

In addition, the author of this thesis authored or co-authored the following publications, which are indirectly related to this thesis in the form of project papers about the RACE project: [36], [318], [72].

CHAPTER 2 Fundamentals

The following section 2.1 introduces definitions of the terminology that we use in this thesis. Section 2.2 shows a basic overview of safety engineering principles and how our approach is resided within these principles. Section 2.3 shows how the approach introduced in this thesis is placed in the field of software and system quality assurance techniques. Section 2.4 provides an overview over architectures and standards in the automotive domain, containing electrical/electronic bordnet architectures and software architectures, like AUTOSAR. Afterwards, section 2.5 gives an introduction about a novel fault-tolerant hardware and software architecture for vehicles, developed in the RACE project. Some properties of the RACE architecture are assumed in this thesis as basic properties of a system, which is aimed to be analyzable with the approach introduced in this thesis. Section 2.6 shows a very brief introduction to architectures and standards in the Avionic domain, like Aircrafts, and briefly discusses degradation concepts in this domain. Finally, section 2.7 gives a brief introduction into different problem solving technologies, like SMT solvers or linear programming (LP) algorithms.

Contents

2.1 Terms and Definitions 7
2.2 Foundations in Safety Engineering 23
2.3 Foundations in Software and System Quality Assurance 24
2.4 Automotive Architectures and Standards 25
2.5 The RACE Approach 30
2.6 Avionic Architectures and Standards 37
2.7 Foundations in Satisfiability Solving and Optimization 38

2.1 Terms and Definitions

In this section, we introduce the main terms and definitions which are related to the topic of this thesis as prerequisite to grasp the domain of this thesis. We focus on an overview over the definitions in the area of dependable embedded systems.

System: A system is a technical product, which is in most cases composed of various parts. Often the parts a grouped to subsystems, which can be designed, constructed and tested standalone. Subsystems can be hierarchically composed of further (sub-)subsystems of various types, such as mechanical, hydraulic, electric (analog) or electronic (digital). An example of a subsystem is a combustion engine block of a vehicle, composed of mechanical parts like piston, crankshaft, camshaft, valves, but also electrical and electronic parts like power lines and electronic control units (ECUs), containing software for fuel injection control, ignition control, and so on. The composition is done by interactions between the subsystems. The interactions are performed through *interfaces*. Each system or subsystem has a *boundary*, at which interfaces exist to interact with other systems or subsystems, or with human users. The electronic subsystems which are not directly recognized by human users are called *embedded systems*. In this thesis, we focus on systems which are composed in a large part of electric and electronic subsystems, like vehicles.

Interface: At the boundary of a system or a subsystem, there exist interfaces, over which nowadays mostly digital data is exchanged. We call the interfaces at the entire system boundary ***external interfaces*** of the system. We call the interfaces between the subsystems ***internal interfaces***. In many embedded systems, physical sensors and actuators establish interfaces to interact with the physical environment of the system. Physical sensors are input interfaces, such as steering wheels or brake pedals, which transform physical information (movements, pressure, temperature, etc.) into digital data. Physical actuators do it vice versa, transforming digital data into physical actions at the output interfaces, such as force feedback at the steering wheel, servo of the steering rack, or hydraulic pump forcing brake piston movements. Also graphical user interfaces (GUIs) exist to interact with human users over displays. In this thesis, we define the system boundary (external interface) to be the sensors and actuators of the system, as well as the digital communication interfaces of the system, allowing to exchange data with other systems (e. g., car-to-x), or also with infrastructure in the Internet (cloud services). All interfaces, regardless of whether mechanical, hydraulic or digital (software), have to be well defined and matching to the interfaces of other systems or subsystems to which interactions are performed.

Behavior: The behavior of a system or subsystem is the transformation of inputs at its input interfaces to outputs at its output interfaces. When the internal transformation procedure is not considered, this is called the black-box behavior or I/O behavior [69]. For the user, the behavior of a system is recognized at the system boundary (external interface). If the system behavior is incorrect (deviation from the specified behavior) at the external interface, then bad things could occur like accidents, or at least not-amused users. If subsystem behavior becomes incorrect at internal interfaces, the question is if and how the entire system behavior is influenced by this at the external interfaces. A system should be able to detect incorrect behavior at internal interfaces and react in such a way that the behavior at the external interface remains correct, even if only in a reduced amount or reduced quality of service (see later introduced terms of *fault-tolerance* and *graceful degradation* in section 2.1.4, as well as *fail-x* in section 2.1.5).

Functional Feature: The behavior at the external interface of a system is grouped to *functional features*. A functional feature is a behavior of a system, which can be used by a human being or another system. Each functional feature realizes at least one of the functional requirements, for which the system is designed for. A functional feature is realized by components of the system, either hardware components like hydraulic, mechanical or electrical elements, or software components that interact with the system environment by sensor/actuator interfaces. In this thesis, we focus on functional features that are realized by software components. Closely related terms to *functional feature* are the terms of *application*, *service* or just *function*, which could be also seen as synonyms somehow. For instance, [23] defines that a *service* delivered by a system is its behavior perceived by users at the system boundary (external interface).

Another kind of features are *non-functional features*, which are features that realize non-functional requirements of the system. Non-functional features ensure so called *non-functional properties*, which do not directly address things that users explicitly expect from the system, but however increase the quality of the system. Some of them are implicitly anticipated by users, like fault-tolerance, others are mostly not perceived by users, like maintainability [178].

Deployment: In this thesis, we use the term *deployment* as the decision about which software components become executed on which hardware execution units (such as micro-controllers, electronic control units (ECUs), etc.). Hence, this decision defines which hardware unit will execute which software components later when the system is put into operation. Beside this software to hardware deployment, there exist also hardware to hardware deployment, like the decision about which sensors or actuators have to be attached to which distributed input/output units in a production plant [211]. A further kind of deployment is the physical deployment of hardware units to a physical location within the system, where the hardware

units have to be placed. Synonyms of the term *deployment* are the terms *allocation* (e. g., used in [156]), *assignment* (e. g., used in [187]), *mapping* (e. g., used in [302]), *binding* (e. g., used in [228]), *distribution* (e. g., used in [155]) and *placement* (e. g., sometimes used in [246]).

However, sometimes the term *deployment* is not understood as introduced above, but instead as the post-production activity [95] of installing software binaries into a system to make them available for use, like continuous redeployment techniques [16] to deploy software updates into a system. Automatic *deployment* plans as sequence of actions to create/delete components and channels in a system are researched in [219]. We do not use the term *deployment* in this manner in this thesis.

2.1.1 Dependability

Dependability: In [217], *dependability* is defined as that property of a computer system such that reliance can justifiably be placed on the service it delivers. In this context, the service that a system delivers is its behavior as it is perceptible by its human or physical users. In [23], the *dependability* of a system is defined as the ability to avoid service failures that are more frequent and more severe than acceptable. Dependability encompasses *attributes*, *threats* and *means*, as shown in Fig. 2.1.

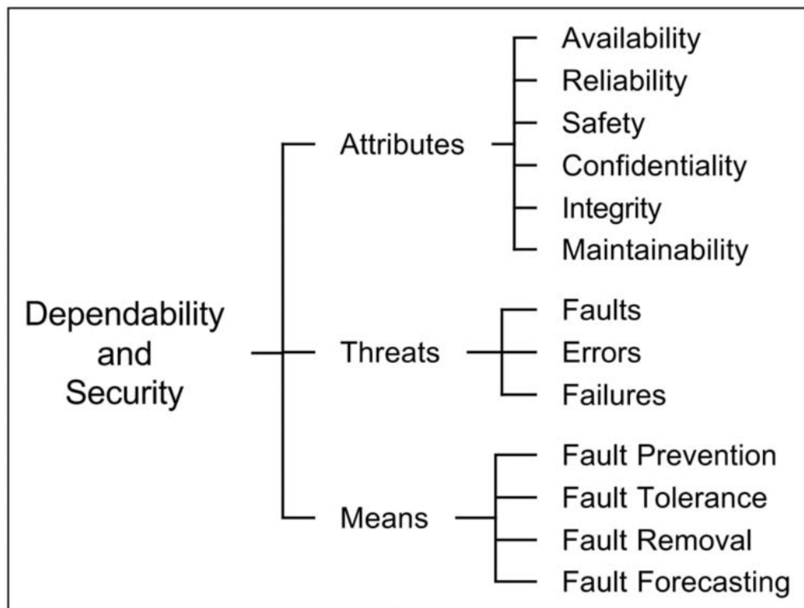


Figure 2.1: Dependability and its tree of attributes, threats and means [23]

The topic of this thesis is related to the mentioned threats; the attributes of availability, reliability and safety; as well as the means of fault tolerance.

Beside the shown dependability tree, other dependability trees are introduced, like the one in [108]. However, we apply the taxonomy of [23] in this thesis.

Security: Dependability and Security are strongly related, as weak security offers possibilities to attack the system and badly influence the dependability. Hence, some of the attributes are also related to security. Security will become more and more important, when vehicles become more and more interconnected [230]. However, we do not further consider security issues in this thesis.

2.1.2 Dependability Threats

Certain threats exist to any system, which may lead to unintended malfunctions and harm. The threats are:

Failure: In [175], failures are defined as termination of the ability of a system element to perform a function as required. Hence, a failure is a transition from correct to incorrect service of a system element or the entire system. Failures are classified in [175] into *common cause failures*, *cascading failures*, *dependent failures*, *independent failures* or *single/dual/multiple-point failures*. Furthermore, *systematic failures* and *random hardware failures* are distinguished in [175]. *Systematic failures* are related in a deterministic way to a certain cause, that can only be eliminated by a change of the design, manufacturing process, or other relevant factors. *Random hardware failures* are defined as failures that can occur unpredictably during the lifetime of a hardware element and that follow a probability distribution. In [23], service failures and dependability failures are distinguished. A *service failure* is an event that occurs when the delivered service deviates from correct service. A *dependability failure* occurs when the given system suffers service failures more frequently or more severely than acceptable.

Beside the above classification, in this thesis we distinguish the following kinds of failures:

1. *External Failure*: a failure of the entire system, recognizable at the systems external interface in form of a behavior that violates the specification.
2. *Internal Failure*: a failure of an element (subsystem) of a system, which does not affect the systems external interface directly. An internal subsystem failure must be handled, such that it does not propagate to the systems external interface and by this becomes an external failure. Hence, we treat an internal subsystem failure as an *error* (defined below) from the perspective of the whole system, which may lead to a failure of the whole system, if not handled appropriately.

Error: An *error* is a discrepancy between a computed, observed or measured value or condition, and the true, specified or theoretically correct value or condition [175]. An error is the part of the total state of the system that may lead to its subsequent service *failure* [23]. In [245], it is defined that an error remains an error as long as it is tolerable and until it becomes intolerable (a *failure*) or it vanishes.

Fault: A *fault* is the cause of an *error* and finally potentially of a *failure*. In [175], faults are defined as abnormal conditions that can cause an element or an item to fail. A fault cannot be directly detected at runtime by a system, only after the fault caused an error, which can be detected. Hence, a faulty state does not yet violate safety [245] and can be seen as tolerable, unless it causes an error. Different types of faults are defined in [23], which can be grouped to be *development faults*, *physical faults* or *interaction faults*. Moreover, there is a distinction between transient faults (appearing once for a short period of time), intermittent faults (aperiodically reappearing for short times) and permanent faults [321].

In this thesis, we focus not directly on faults, but on errors and failures that may either be caused by physical faults, which include all fault classes that affect hardware, or caused by systematic development faults of software components. We consider errors caused by intermittent or permanent faults being permanently handled by applying an isolation of the faulty system element from the residual system. The system may handle transient faults either also permanently by isolation, or by masking the once appearing erroneous value for instance by using a valid *last good known* value for a short period of time. We focus on the cases in which system elements become isolated permanently.

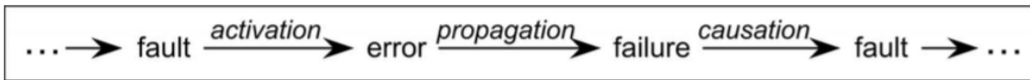


Figure 2.2: Chains of threats, from Fig. 11 in [23]

Fig. 2.2 shows how the introduced terms of faults, errors and failures are related to each other [23]. A fault is the cause of an error and an error may lead to a subsequent failure. However, the perspective onto the considered system element is important. As described above, an internal failure of a sub-element of a system is not equal to an external failure of the entire system. But a sub-element failure can be seen as entire system fault leading to an entire system error, which shall be detected and handled therewith the local sub-element failure does not propagate through other parts of the system, leading to an entire system failure.

In this thesis, we consider situations in which execution units of a system have random hardware failures, which got detected by an appropriate mechanism, followed by an isolation of the failed execution unit. Hence, an execution unit failure is an internal failure. It shall not lead to an external failure, but may lead to a reduction of the functionality recognizable at the external interface. This reduction of functionality is called degradation, respectively graceful degradation, introduced below in section 2.1.4.

With more focus on purely software based products, the above terminology is extended in [351] and [350] with the terms of

- Mistake: A mistake is a human action that produces a fault.
- Defect: Defects are the superset of faults and failures.

Related to the above terms are the following terms, all introduced in [175].

- Harm: Physical injury or damage to the health of persons.
- Hazard: Potential source of harm caused by malfunctioning behavior of an item. An item is a (sub-)system or array of (sub-)systems to implement a function at the entire vehicle level.
- Severity: Estimate of the extent of harm to one or more individuals that can occur in a potentially hazardous situation.
- Risk: Combination of the probability of occurrence of harm and the *severity* of that harm. In other words, risk is characterized by the two properties of frequency of hazardous events and the severity of hazardous events.
- Malfunctioning behavior: Failure or unintended behavior of a system or subsystem with respect to its design intent. This means, a functional feature of the system deviates from its specification. Hence, the behavior is wrong at the external system boundary.
- Failure Mode: Manner in which the hardware or software of a system or subsystem fails. In [264] defined as the manner by which a failure is observed, generally describing the way the failure occurs and its impact on equipment operation.
- Failure Effect: consequence(s) that a *failure mode* has on the operation, function, or status of a system or subsystem [264].

2.1.3 Dependability Attributes

We focus on the attributes, which are related to the topic of this thesis.

Safety: *Safety* denotes the ability of a software system to avoid failures that will result in loss of life, injury, significant damage of property, or destruction of property [330]. In [23], safety is defined similarly as absence of catastrophic consequences on the users and the environment. The ISO 26262 defines **functional safety** as the absence of unreasonable risk due to hazards caused by malfunctioning behavior of electric/electronic systems [175]. In context of autonomous driving, [54] introduces **operational safety** as property or ability of a vehicle to be in a suitable operating condition or meeting acceptable standards for safe driving and transport. They define *operational safety* (also called *roadworthiness*) as a composition of the dependability attributes of safety, availability, reliability and maintainability.

Reliability: The *reliability* of a system or subsystem is defined as its continuity of correct service [23], or as measure of the ability to work completely failure-free for a certain length of time, or mission [207]. In [88], reliability is defined as the ability of a system to operate correctly according to its specification for a given time interval. [330] defines a software systems *reliability* as the probability that the system will perform its intended functionality under specific design limits, without failure, over a given time period.

In this thesis, we associate *reliability* to the required level of fail-operationality of functional features (see also definition of fail-operational in section 2.1.5). The higher the required fail-operational level is, the higher is the required reliability of a feature. As different features might have different fail-operational levels, we obtain a *mixed reliability* system.

Availability: A software systems *availability* is the probability that the system is operational at a particular time [330]. It is also defined as *readiness for correct service* [23]. An overview over more definitions of *availability*, as well as an approach to specify and analyze the *availability* of software intensive systems, is given in [186], mentioning that *availability* is defined as the ability of a system to operate without failure most of the time.

To obtain a quantitative metric for the availability, often the following definitions are used

- Mean time to failure (MTTF): average time from startup to a failure of a system. Typically used for systems, which cannot be repaired.
- Mean time between failures (MTBF): average time between two consecutive failures of a system. Typically used for systems, which can be repaired.
- Mean time to repair (MTTR): average time required to repair a failed system.

The *availability* A is defined as

$$A = \frac{MTTF}{(MTTF + MTTR)} \quad [145]$$

or

$$A = \frac{MTBF}{(MTBF + MTTR)} \quad [212][144]$$

In the latter definition of [212] [144], it is assumed that MTBF contains only the time in which a system is available (average uptime), but does not contain the repair time MTTR. Other approaches consider $MTBF = MTTF + MTTR$ [117], what leads availability to become

$$A = \frac{(MTBF - MTTR)}{MTBF} \quad [117]$$

To handle this clash between two different definitions of MTBF, a *mean time between down event* (MTBDE) has been introduced for repairable systems as sum of the average uptime (MTBF) and the average downtime (MTTR), $MTBDE = MTBF + MTTR$.[1]

However, in this thesis we only consider a boolean $\{0,1\}$ availability metric of functional features for certain points in time. As the formal model that we are going to introduce has no notion of time intervals, we do not consider the availability over a time interval, like the above mentioned availability metrics do it. Instead we allow to analyze a given situation like a degradation scenario, with respect to if a specific functional feature of a system can be kept available or if it has to be deactivated.

2.1.4 Dependability Means

Fault-tolerance: Fault-tolerance is the ability of a controlled system to maintain control objectives, despite the occurrence of a fault [53]. According to [330], a software system is fault-tolerant, if it is able to respond gracefully to failures.

In [245], *masking* and *non-masking* fault-tolerance are distinguished. While masking fault-tolerant systems mask the effects of faults (namely errors and failures) such that the effects are hidden from users, non-masking fault-tolerant systems do not hide the effects of faults from users and for instance reduced availability may be recognized. This is also related to static and dynamic redundancy, what we introduce later in section 2.1.6.

In [285], fault-tolerant systems are defined to be able to keep the system running at 100% of its designed functionality even in case of failures, while *self-healing* systems (cf. section 3.1.6) may operate with less than 100% functionality after a healing procedure. However, contradicting to this definition of [285], in [53] it is defined that degradation of control performance may be accepted for fault-tolerant systems. In this thesis, we define fault-tolerant systems to be able to apply graceful degradation, meaning that a reduced level of provided functional features can be acceptable in failure scenarios, but this has to follow certain constraints that we discuss later in chapter 4.

Graceful Degradation: In [300], graceful degradation is defined as a smooth change of some distinct system feature to a lower state as a response to an event that prevents the system from exhibiting that feature in its full state, often used to allow systems to survive errors or internal failures by removing their damaged parts. Graceful degradation is classified in the dependability means (cf. Fig. 2.1) as an error mitigation mechanism for error correction to establish fault tolerance [300], see Fig. 2.3.

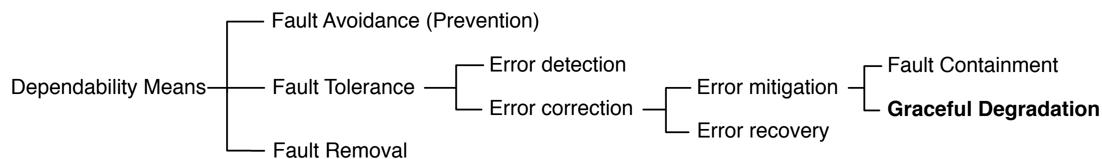


Figure 2.3: Graceful Degradation classification into dependability means [300]

The degradation can be applied for instance in case of an internal failure of a system element. This means, graceful degradation is a concept to tolerate failures by reducing the functionality or the performance, rather than shutting down the system completely [313]. *Performance degradation* means to keep

[1] `http://www.weibull.com/hotwire/issue94/relbasics94.htm`, last access April 5th, 2016

alive the same functionality, but with reduced performance. *Functional degradation* denotes a degradation of a system in such a manner that it continues to operate, but provides a reduced level of functionality rather than failing completely [280] [287]. In this thesis, we focus on functional degradation on a structural level.

Furthermore, beside the preferred initial behavior, gracefully degrading systems permit additional weakly consistent behaviors which are undesired, but tolerated and sufficiently close to the preferred behavior [163]. However, in this thesis we do not consider such behavior degradation, as the formal model that we are going to introduce has no notion of behavior of functional features or software components.

In [287], graceful degradation in *timeliness* and graceful degradation in *quality* is distinguished. Timeliness is the ability of a service to perform its required functions and provide its required responses within specified time limits. A degradation of timeliness is similar to performance degradation. Quality degradation is a degradation of a services correctness and/or how usable a service is [287], what is similar to functional degradation, to which we focus on.

Another definition is that graceful degradation is a resilient system's ability to survive disruptions originating from within or without while still carrying out its missions [91]. The ISO 26262 defines *degradation* as strategy for providing safety by design after the occurrence of failures [175] and that *graceful degradation* at the software level refers to prioritizing functions to minimize the adverse effects of potential failures on functional safety [176].

In [300], three patterns are introduced about how to smootly reach a lower system state. They distinguish an 1) *optimistic*, 2) a *pessimistic* and 3) a *causal* degradation pattern.

1) optimistic degradation pattern: remove only the failed system element
2) pessimistic degradation pattern: remove all system elements that are anyhow related to the failed element
3) causal degradation pattern: remove all system elements that strongly depend on the failed element

With respect to the patterns introduced in [300], in this thesis we consider degradations in form of a mixture of pessimistic and causal degradations, as we distinguish optional and mandatory communication channels between software components. If optional input data becomes unavailable, the receiving component can continue to operate. If mandatory input data becomes unavailable, the receiving component becomes deactivated (causal). However, we have no internal whitebox data-flow consideration of components, leading to the deactivation of the whole component if one single mandatory input data item is missing, although the residual input data might be sufficient to fulfill a subset of the specification (pessimistic).

In this thesis, we define *graceful degradation* as follows:

> A degradation is graceful, if no fail-operational requirements are violated by a degradation, and if mixed-criticality functional features become deactivated sequentially by starting with the deactivation of the feature with the lowest criticality.

We focus on degradations that become necessary due to assumed failures of execution units or software components, forcing loss of non redundant components, or requiring deactivations of components due to insufficiency of input data or insufficiency of execution resources.

However, because we do not model the functional behavior of components by means of Input/Output data stream relations at component interfaces, as introduced for instance in [69], our approach does not include a notion of a *Quality of Service* (QoS) of functional behavior. This means, we do not analyze to which sense users do experience how the system behavior is influenced by degradations, and to which sense this is accepted by users as graceful or not.

2.1.5 Other Definitions related to Dependability

Beside the above three dependability attributes, which are listed in [23], there exist certain more terms in the field of dependable systems, listed below.

Safety Case: Argument that the safety requirements for an item are complete and satisfied by evidence compiled from work products of the safety activities during development [175].

Resilience: Resilience is the ability of a system to provide and maintain an acceptable level of service in the face of various faults and challenges to normal operation.[2] Resilience can be considered as the flip side of vulnerability [86]. In [98], *dynamic resilience* is introduced as a system's capacity to respond dynamically by adaptation in order to maintain an acceptable level of service in the presence of impairments. They introduce *predictable dynamic resilience* as the capacity of a system to deliver dynamic resilience within bounds that can be predicted at design time. In this thesis, we contribute to analyze the predictable dynamic resilience of the system under analysis at design time.

Survivability: Survivability is a software systems ability to resist, recognize, recover from, and adapt to mission-compromising threats [330]. In other words, it is the capability of a system to fulfill its mission, in a timely manner, in the presence of attacks, failures, or accidents [112]. Survivability is very often seen in connection to graceful degradation. In [202], survivability is defined as how systems will degrade functionality in the presence of failures, as it is the degree to which essential functions are still available even though some part of the system is down [96].

Fault tolerance is often seen as a subset of survivability[3], and survivability is often seen as a subset of resilience.[4]

Robustness: A software system is robust, if it is able to respond adequately to unanticipated run time conditions [330]. Robustness is usually associated with reliability and resilience [153].

Fail-x: There exist several ways on how to react to a failure at runtime. Below follows an introduction to the main concepts and terms, related to this thesis.

- Fail-Silent: A *fail-silent* node is a self-checking node that either functions correctly or stops functioning after an *internal* failure is detected [64]. Important here is the *internal* failure, which means that the node itself has not yet fail behavior at the external node interface, but a failure of an internal sub-element of the node is detected. An example is a micro-controller node that detects a data distortion failure of its internal random-access memory. A fail-silent node avoids that the internal failure becomes an external failure of that node, propagating to other nodes of the system. Instead, a fail-silent node stops sending actual data to other nodes or actuators in the occurrence of an internal failure. This avoids propagation of faulty data to the rest of the system. Also in [182], it is defined that after one (or several) failure(s), a fail-silent component exhibits quiet behavior externally (i.e., stays passive by switching off) and therefore does not wrongly influence other components. However, the state of the system at that moment in which it goes into fail-silent mode may be not safe. For instance, if the last outgoing data was forcing an actuator to accelerate, the actuator receives no data telling him that it shall stop to accelerate.

[2] `https://wiki.ittc.ku.edu/resilinets/Definitions#Resilience`
[3] `https://wiki.ittc.ku.edu/resilinets/Definitions#Fault_Tolerance`
[4] `https://wiki.ittc.ku.edu/resilinets/Definitions#Survivability`

- Fail-Safe: In case of an internal failure, a *fail-safe* system switches to a state that is considered safe in the particular context. No harm is caused in case of a failure. In [182], it is defined that after one (or several) failure(s), a fail-safe component directly reaches a safe state (passive fail-safe, without external power) or is brought to a safe state by a special action (active fail-safe, with external power). In the example of the acceleration actuator, this means that a fail-safe system ensures that a safe state is entered, meaning the actuator to stop the acceleration and most probably to stop the movement completely (but this depends on the system properties).

- Fail-Operational: In [53], a system is defined to be *fail-operational*, if it is able to operate with no change in objectives or performance despite of any single failure. This means, the system has to be able to tolerate at least one failure of a sub-component (internal failure) and stay operational without going into a fail-safe state and without suffer an external failure. For instance, fail-operational is required if no safe state exists immediately after a system component fails [182].

  In this thesis, we consider fail-operationality not dedicated to an entire system, but to the single functional features of the system. Hence, some functional features may be fail-operational, others not. Furthermore, we distinguish different levels of fail-operationality. A functional feature with a fail-operational level of x is required to continue operation after the first x failures of hardware or software components in the system. Afterwards, the feature is allowed to become disabled. In this thesis, we consider random hardware failures of execution units and systematic failures of software components as occurring failure types that have to be handled by the system to enable fail-operational features.

- Fail-Degraded: In this thesis, we consider systems in which only a subset of the functional features is required to behave fail-operational. However, the other features are allowed to be deactivated in case of failures of subsystems. This means that not the entire system is fail-operational, as some features might become deactivated, resulting in a degradation of the system. Such systems are called *fail-degraded* systems, like in [34].

Certain more terms exist in context to the above terms, like fail-passive, fail-secure, fail-fast, fail-halt, fail-stop, which we do not further introduce here.

In the context of Steer-by-Wire systems, a study of the Daimler AG [122] has shown that by fail-silent designed Steer-by-Wire systems, no functionalities with customer benefit can be realized, as already small changes in the steering ratio respectively small steering angle faults cannot be handled by the drivers and result in problems. A mechanical backup guarantees in that context no transition to the safe state. The conclusion of Daimler investigations is that an appropriate Steer-by-Wire system has to be designed *fail-tolerant* to make sure that the steering system will change directly to the safe state when failures occur (fail-safe). Thus Steer-by-Wire makes noticeable higher demands to the system concept as known driving dynamic systems (like anti-lock braking system (ABS) or electronic stability program (ESP)) and even Brake-by-Wire.

Real-Time Scheduling: A real-time computer system is a computer system, in which the correctness of the system behavior depends not only on the logical results of the computation, but also on the physical time when these results are produced [209]. This means, the timing has a huge impact on the dependability of a system. Real-time systems are categorized to be hard or soft, static or dynamic, preemptive or non-preemptive [209]. The latest time at which a task has to be finished is called the *deadline*. In hard real-time systems, a task which misses its deadline may cause a catastrophic failure. Think about an airbag which opens too late. However, hard real-time systems are not defined to be *fast* or *performant*, but instead they are defined to be *predictable* [74].

In this thesis, we assume a real-time system to use *logical execution times* (LETs), which abstracts from the actual execution time of a program on a physical implementation platform [200]. We assume a LET based model of computation, in which software components are scheduled in execution cycles of fixed cycle length. Input data for all components is captured at the beginning of the execution cycle and not modified anymore during the cycle, output data of all components is collected at the end of each cycle, not before. See section 2.5.1 for more details about this.

Criticality Levels: For the automotive domain, the ISO 26262 [174] defines different *Automotive Safety Integrity Levels* (ASIL) from QM (Quality Management) over ASIL A, ASIL B, ASIL C to ASIL D, where ASIL D denotes the highest criticality (see also section 2.4.3). In the Avionic domain, the DO-178B/DO-178C [288] defines *Design Assurance Levels* (DAL) from DAL E (no safety effect) over DAL D (minor), DAL C (major), DAL B (hazardous) to DAL A (catastrophic). Hence, without looking into details, the ISO 26262 level QM is comparable with DO-178C *DAL E*, and *ASIL D* is comparable with *DAL A*.

A system or subsystem is called to be ***mission critical***, if a failure might prevent an operation or task from being performed, possibly preventing successful completion of the operation as a whole [284]. Mission criticality corresponds typically to the lower levels of the standards, like DAL D or DAL C (minor or major) for the avionic domain [284]. Typical examples of mission critical features are navigation/display or mission command and control subsystems.

A system or subsystem is called to be ***safety critical***, if a failure could result in danger, injury or loss of human life [284]. Safety criticality corresponds typically to the higher levels of the standards, like DAL B or DAL A (hazardous or catastrophic) for the avionic domain [284], like flight and engine controls.

Mixed Criticality: A *mixed criticality* system has multiple distinct criticality levels [73], like the levels introduced above. This means, such a system provides functional features – and contains associated subsystems – that have different criticality levels. Those subsystems that have different criticality levels have to be separated from each other to avoid negative influence from less critical subsystems to high critical subsystems. One approach to handle different criticalities is to avoid interference by physical separation into multiple physical devices with clearly separated network connections.

However, if mixed criticality software components are executed by a single control unit, we talk about mixed criticality controllers. Mixed-criticality embedded controllers are gaining attention for instance in the automotive and avionics domain due to savings in cost, space, weight, heat and power [73] [261]. Also in [131] it is stated that future automotive E/E-architectures will consist of highly integrated domain-controllers, providing very high functional integration. This requires the support of the execution of application components with mixed criticality levels on the same controller [70]. *Separation* of the mixed criticality software on mixed criticality controllers is required, as introduced below.

Separation by Partitioning on Mixed Criticality Controllers: Having mixed criticality controllers, the mixed critical software components have to be separated from each other to avoid undesired influences between the different components. For this, mechanisms for spatial and temporal partitioning are required [292] (also called time and space-partitioning [261]), as offered by so called separation kernel operating systems [294], such as *PikeOS*[5] [190] or *EB tresos Safety*[6]. This ensures that 1) *spatial influence* by memory access to other components is avoided, as well as 2) *temporal influence* by deferring the execution of other components in the schedule is avoided.

Fault Containment Regions and their Isolation: Isolation is the mechanism to detach a system element that has an error or a failure from the residual system in order to avoid an entire system failure. To be

[5] `https://www.sysgo.com/products/pikeos-hypervisor`

[6] `https://www.elektrobit.com/products/ecu/eb-tresos/functional-safety`

able to isolate system elements in a spatial and temporal manner, systems are often partitioned into independent so called *fault containment regions* (FCR) (e.g., [171]). Errors that appear within an FCR must be detected and isolated by error-detection mechanisms at the boundaries of the FCR such that the errors cannot damage the computational state in any other FCR. Hence, a containment region defines the border where fault propagation must stop [283]. The aim is that no single faulty FCR can knock out the whole safety-critical system [208]. Hence, isolations deal to achieve fault containment, as required for safety assurance [148].

As errors or failures have to be detected before they can be isolated, the procedure is also called *fault detection and isolation* (FDI) [168], although it should be called error- or failure-detection according to our terminology.

Recovery: Recovery is one form to ensure *reliability* [233]. Distinguished can be *error recovery*, enabled by recovery actions performed after an error has been detected [172], and *failure recovery* to recover from already occurred failures. A recovery action is performed to remove an error or a failure, or at least isolate the respective entity to avoid further propagation. In [102], *proactive recovery* is defined as periodically initialize replicas with a correct application state all non-faulty replicas have agreed on. This is useful to handle byzantine-faulty replicas by recovering replicas periodically independent of any failure detection mechanism [76]. We do not consider proactive recovery in this thesis. Instead, we consider *reactive recovery*, applied in fault-tolerant systems to handle detected errors. We refer to [265] for a further discussion about proactive and reactive recovery. One form of a reactive recovery action is a failover, see below.

Failover: A *failover* is a switch from an until now active but failed system element to a backup element that takes over the functionality. Without the failover mechanism, the functionality provided by the failed element would be either gone (if fail-safe or fail-silent), or even worse the system would run out of control. Hence, the failover mechanism is important to establish fail-operational behavior. A failover is performed automatically by the system, which is the difference to a *switchover* requiring manual user-interaction. In this thesis, we consider failovers between redundant identical or diverse software components in case of scenarios of failed hardware or software.

2.1.6 Redundancy and Replication Mechanisms

Redundancy and Replication: To increase certain properties of the system, like fault-tolerance, reliability, availability or accessibility, the mechanisms of redundancy and replication have been introduced.

- *Redundancy* is the duplication of (mostly critical) components or functions of a system, usually in form of backup components. It is about providing multiple (often identical) instances of the same system element and switching to one of the remaining instances in case of a failure (failover).[7] This means, only one copy is in operation and provides the intended service. The backups are not in operation, until one of them becomes required. Hence, redundancy permits a product to operate even though certain parts and interconnections have failed, thus increasing its reliability and availability [320]. There exists a tradeoff between redundancy and cost [267] [268] and also between redundancy and reliability [325].

- *Replication* is about providing multiple (often identical) instances of the same system or subsystem, directing tasks or requests to all of them in parallel, and choosing the correct result on the basis of a quorum.[7]

[7] `https://en.wikipedia.org/wiki/Fault_tolerance#Replication`, accessed at 13th August 2015

Different types of replication are distinguished, like

- *Active Replication*: all replicas are active and calculate results simultaneously [28]. A voter selects the predominant output and ignores faulty replicas.
- *Passive Replication*: a passive replica does not contribute to provide any functional feature [28]. The state and result of the active replica is synchronized to the passive replicas at certain points in time. In case the active replica becomes faulty, it shuts down in a fail-silent manner and one passive replica becomes active. One example for passive replication is the so called *Primary-Backup Replication* [221].
- *Leader/Follower Replication*: one active leader exists, as well as multiple active follower replicas [28]. In case the leader replica fails, one follower becomes the new leader. No voting is required.

An example of a system using active and passive replicas is shown in [102].

There exist different implementations of redundancy, like information redundancy, time/temporal redundancy, software component or hardware component redundancy. An additional distinction is done between functional and physical redundancy [91].

Popular examples of hardware component redundancy are:

- *Dual-Modular Redundancy* (DMR): DMR is a form of active replication. One example of DMR is a *lockstep* system that runs the same set of operations at the same time in parallel, e.g. lockstep processors with two cores working totally bit-identical. For DMR, a two-way *comparison* subsequent logic is needed to detect discrepancies in the outputs, followed by shutting down or isolating the DMR system (fail-silent). E.g., if a hardware failure occurs in one core of a lockstep processor, the output of the two cores become different, what is detected by the comparison mechanism and handled by switching off the whole lockstep processor (as it cannot be determined which core failed). Synonyms for DMR are *Loosely-Synchronized Dual Processor Architecture* [25] and *Self-Checking Pair* [154].
- *Triple-Modular Redundancy* (TMR): Also TMR is a form of active replication. It involves the use of three subcomponents (or "modules") of identical design, and majority voting circuits which check the module outputs for exact equality. It is thus designed to mask the failure of any single module, by accepting any output that at least two of the modules agree on [280]. TMR is also called *fixed replication* with voting in [280]. While the structure is fixed (fixed replication, static redundancy, see Fig. 2.5 below), the replicas are executed and active (active replication). Another synonym for TMR is *2-out-of-3 (2oo3) Architecture* [57] [203].

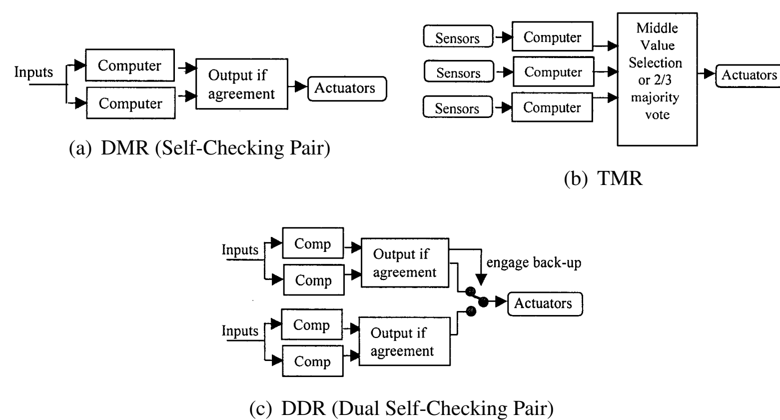
  For TMR, a three-way *voting* subsequent logic is needed for selecting the output result that should finally be given to the receivers. It has to be determined, which of the redundantly calculated values should be taken (in case 1 of the 3 components fails) (fail-operational). As the voting is also a critical task, the voting logic has to be redundant, too.

  An extended form of TMR, using two hierarchical levels of TMR (called triple-triple redundancy), is for instance applied in Boing 777 airplane [358].
- *Duo-Duplex Redundancy* (DDR): A duo-duplex architecture consists of two dual-modular redundancy (DMR) subsystems in parallel. Hence, overall four components exist, two in each DMR subsystem. One DMR subsystem is active and the other one is passive [17]. Both DMR subsystems check themself for correctness by internally comparing data of the two contained controllers [286].

When a failure is detected in the active DMR subsystem, a failover is performed. The failed DMR subsystem becomes isolated (fail-silent) and the passive DMR subsystem becomes active. DDR is a form of passive replication, as the second DMR subsystem only becomes active when the first DMR subsystem fails.

In total, a *fail-operational* behavior is reached. Like in TMR, only one fault can be tolerated [182]. After the failover, either the full functionality can be kept alive (fully fail-operational), or a degraded version of the functionality (fail-degraded, often called *limp-home* backup [203]). Synonyms for DDR are *Dual Lock-Step Architecture* [25], *2-out-of-2 Diagnosis-Fail-Safe (DFS) Architecture* [203] and *Dual Self-Checking Pair* [154].

Fig. 2.4 shows illustrations for the three mentioned hardware component redundancy types.



(a) DMR (Self-Checking Pair)

(b) TMR

(c) DDR (Dual Self-Checking Pair)

Figure 2.4: Illustrations for DMR, TMR and DDR from [154]

Also other combinations beside classical DMR, TMR or DDR exist. For instance, a patent for a system of two lanes, each lane having a primary and a redundant processor, and subsequent three different monitors plus one selection logic is given in [157].

In [275], a *roll-forward checkpointing* mechanism is introduced, comprising a primary DMR subsystem and a secondary spare DMR subsystem, latter instantiated only on-demand if an inconsistency is detected between the two lanes of the primary DMR. The spare DMR re-executes the inconsistent command sequence to identify which of the two primary DMR lanes was erroneous. Due to this, the meanwhile continuing primary DMR subsystem can be corrected by state transfer, without requiring a *rollback recovery* to the last valid checkpoint. This is helpful to ensure predictable execution times in real-time systems. Recovery by checkpointing is also considered in [143].

As it might be already noticed, the terms of redundancy and replication are closely related to each other and cannot be clearly separated. One example for using the terms in relation is that an example of *active replication* is *triple modular redundancy* (TMR) [130], or that redundancy typically refers to replicated hardware and/or software within systems [91].

Fig. 2.5 shows an overview about additional characterizations of redundancy. Some areas in the figure are relevant in the context of this thesis.

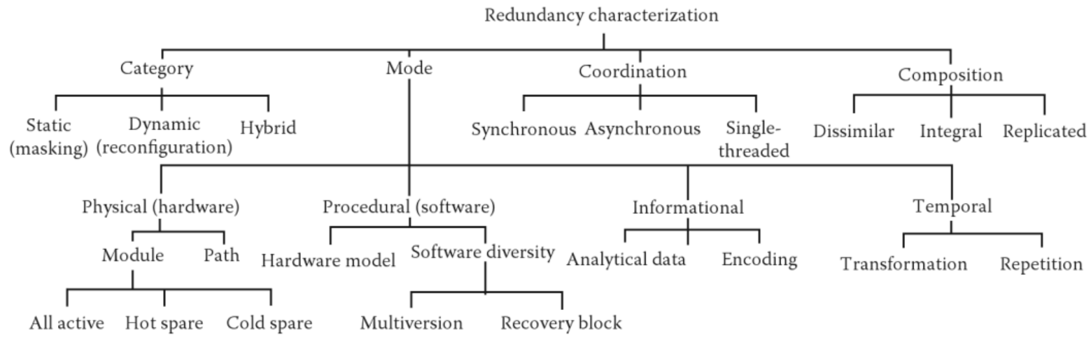


Figure 2.5: Redundancy characterization from [321] (Fig. 5.5)

First, Fig. 2.5 shows in the upper left part the redundancy categories of *static* (masking) and *dynamic* (reconfiguration) redundancy, as well as hybrids of it.

- *Static redundancy* is to use redundancy to mask or hide the effects of faults in a component [280]. It is also called *masking redundancy* (in [280]) or *passive redundancy*.[8] No active action is required by the system, like activations of backup components. Examples of static redundancy are active replication with voting, like TMR, but also *N-version programming* [80].
- *Dynamic redundancy* is to use active recovery techniques like reconfiguration or self-healing. It is also called *active redundancy*.[8] Examples are systems using standby spares replacing failed units [280]. More about standby spares is introduced below.

Please notice that with both static (masking) and dynamic redundancy, masking fault-tolerance can be achieved (cf. definition of fault-tolerance in section 2.1.4).

Second, in the lower left part of Fig. 2.5, physical module redundancy types are listed. These are (plus additional warm spare):

- All active: This refers to active replication, as defined above. For instance, in a TMR system, all three copies are active.
- Hot spare: A hot spare is active, but does not produce output signals. Hence, it does not contribute to provide any functional feature. The internal state of a hot spare is updated to the states of the active replicas continuously. This allows a very low MTTR by a very fast failover, to let the hot spare become active. In [105], a hot spare is defined as an element whose degradation and failure behavior - while it is a spare - is the same as while it is active. This is because the only difference between an active replica and a hot spare replica is that the latter does not produce outputs. A hot spare concept for hardware is for instance applied in aircrafts [358].
- Cold spare: A cold spare is a passive replica. The replica is normally not executed in schedule, but the binary is just ready to be started. Hence, the internal state is not updated to the state of the active replicas during normal operation. When the cold spare has to become active (failover), it starts with its initial state, which may be different to the last state of the failing replica, and potentially required resources have to be acquired during startup. This means, the failover is slow and the MTTR is high. But as the replica is passive, less resources are required during normal operation, compared with

[8] `http://www2.cs.uidaho.edu/~krings/CS449/Notes.S13/449-13-03.pdf`, see slide 8

using a hot spare. In [105], a cold spare is defined as an element that neither degrades nor fails while it is a spare. Cols spares require dynamic redundancy and can also be implemented with diversity, like used in the concept of *recovery blocks* [172].

- Warm spare: In addition, there exist the term of warm spares, being replicas which are already started and then passivated (like cold spares), but in some points in time they get information about the current internal state of the active replicas (like done for hot spares). This is similar to rollback to a *checkpoint* [367] [274]. The MTTR is medium, between the MTTR of cold spare and hot spare.[9] In [105], a warm spare is defined as an element whose degradation and failure behavior lies between a cold spare and a hot spare.

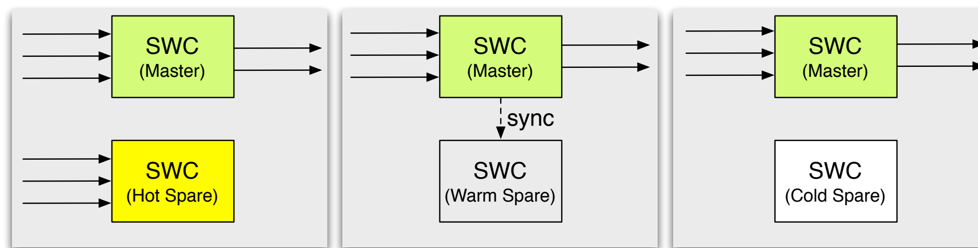


Figure 2.6: Active I/Os of hot, warm and cold spares

Fig. 2.6 shows the activity of input and output channels for hot-, warm and cold spares of software components (SWCs), as we use it in this thesis combined with a primary master instance. We consider spare components not only for physical hardware components, but particularly also for software components. A hot spare gets inputs, but its outputs are ignored. A warm spare becomes synced with the master at certain checkpoints in time. A cold spare is not executed in schedule and hence, it gets no inputs.

A synonym for a *spare* component is a *standby*, *slave* or *backup* component. One system using cold, warm and hot standbys is the QNX Neutrino operating system.[9] Another example of a system using hot spares is a hybrid quad redundant scheme comprising four controllers, in which three of the four controllers are active replicas, running as a TMR subsystem, and the fourth controller is a hot spare which only becomes active in case one of the other three controllers fails, in order to substitute it (see [321], Fig. 7.13, page 7-16). If then afterwards one further controller fails, a DMR subsystem is left.

Diversity: Third, Fig. 2.5 shows *software diversity*, which is another important concept in the context of redundancy and replication.

- *Diversity*: Diversity is to provide different implementations of the same specification. The different implementations are then replicated, to cope with systematic errors in a single implementation. This avoids systematic errors in all replicas, like bugs encoded in software. Diversity can be given at software or hardware level. Examples of concepts providing software diversity are *N-version (Multiversion) programming* [80] and *recovery blocks* [172]. Examples of diverse redundancy on hardware level are different sensors for the same value, like temperature sensors with negative and positive temperature coefficients, attached to the same physical location. A survey about more different kinds of software diversity is given in [32].

[9] http://www.qnx.com/developers/docs/660/topic/com.qnx.doc.neutrino.cookbook/topic/s1_ha_Standby.html

- *Dissimilarity*: A term closely related to diversity is dissimilarity. In context of a TMR system in the avionic domain, it is for instance stated that *dissimilarity* needs to be judiciously used for the program risk reduction and will not be an alternate to the rigorous verification and validation analysis/testing activities [358].

Hence, homogeneous redundancy (without diversity) and heterogeneous redundancy (with diversity) is distinguished [104]. Redundancy respectively replication mechanisms and patterns are also used or introduced for instance in [109], [251], [160], [185], [274] and [352]. For more information about the residual terms in Fig. 2.5, we refer to [321].

2.2 Foundations in Safety Engineering

The overall goal of safety engineering is to ensure *freedom of unacceptable risk* [336]. The basic steps in proving the safety of a system are shown in Fig. 2.7. The terms are commonly used in the automotive domain.

Figure 2.7: Safety engineering lifecycle from [336]

The safety analysis approaches aim to analyze the possible effects of faults in a system. The most relevant approaches are:

Fault Tree Analysis (FTA): FTA provides a logical method for graphically presenting the chain of events leading to a system failure, determining system safety and reliability from the event probabilities [218]. FTA is a deductive top-down method [60] considering combinations of events in the cause-path to analyze their effects on a system and by this to understand how a system can fail. FTA is for instance applied in [344] and [242]. Specialized versions of FTA exist, like *State Event FTA* [188] or *Dynamic FTA* [105]. The latter supports redundant spare backup components like cold-, warm- and hot-spares, as we introduced them in section 2.1.

Failure Mode and Effects Analysis (FMEA): FMEA is a systematic way of identifying failure modes of a system, item or function, and evaluating the effects of the failure modes on the higher level [264]. FMEA is an inductive bottom-up analysis [60] of the effects of single element failures onto subsystems and systems. A failure mode is defined as a particular way in which an item fails, independent of the reason for failure [249]. Also approaches for probabilistic FMEA exist [147] [10], based on applying *Markov chains*.

System-Theoretic Process Analysis (STPA): STPA is a modern hazard and safety analysis technique, based on the accident causation model *System-Theoretic Accident Model and Process* (STAMP) [223], applicable early in the design process of a system to achieve an acceptable risk level [3]. The aim is to identify the potential hazardous causes in complex safety-critical systems at different architecture abstraction levels [1]. To support software-intensive systems, the STPA SwISs approach exists to combine safety analysis and software test case generation [1] [216]. An automotive case study of applying STPA SwISs is shown in [2]. With A-STPA respectively its successor XSTAMPP [4], an open-source tool platform exists for the STAMP/STPA approach.

This thesis: The approach introduced in this thesis aims to support FMEA approaches with arguments towards reliability by guaranteeing the achievement of fail-operational requirements. For mixed-reliability systems, we also analyze necessary degradations in failure scenarios due to insufficient resources, caused by failures. We do not distinguish different failure modes (ways in which a system element can fail), but we assume the presence of a mechanism to detect any kind of failure mode of certain system elements, and to isolate the failed elements from the residual system to avoid propagation. Our approach provides a formal structural effect analysis of the decreasing resources due to such isolations. We combine this with the synthesis of valid redundant deployments and incorporated failover scenarios to ensure the fulfillment of all fail-operational requirements of functional features of the system under analysis, if feasible. If required due to insufficient resources in failure scenarios, we synthesize a valid system degradation by partially deactivating features having no fail-operational requirement. As example for a redundancy and failover mechanism, we assume a present safety and fault-tolerance concept that has been developed in context of the RACE project (see section 2.5.2). With respect to the fault-tolerance concept, we verify that all fail-operational requirements can be met in all considered failure scenarios. This supports to establish further evidence for required reliability of functional features of the system under analysis.

2.3 Foundations in Software and System Quality Assurance

Many approaches were proposed to tackle quality assurance in order to obtain high quality software and systems. These approaches can be categorized into *constructive approaches* and *analytic approaches* [350], as well as *process based approaches* [193]. Constructive approaches are for instance the use of well approved architecture or design patterns or architectural styles. Process based approaches are the development according to standards or models like ISO 9000 or Automotive SPICE. The analytic approaches are distinguished into dynamic analysis and static analysis. Dynamic analysis requires to execute parts of the implemented system, like unit-testing or runtime fault-injection and monitoring. Static analysis does not require to execute an implemented system. This includes for instance human code reviews or walkthroughs, as well as formal methods like static program analysis based on *theorem proving* (e. g., [90]), *data-flow analysis* or *symbolic execution* (e. g., [198]).

The approach which is introduced in this thesis is a static system analysis, applying a formal verification, based on using an SMT solver. The input to our approach is a model of the systems hardware and software architecture enriched with relevant properties, like requirements to behave fail-operational. As the development of automotive embedded systems and software is increasingly realized using model-based design [68], these models can be partially used as basis for our input models.

2.4 Automotive Architectures and Standards

2.4.1 Classical Automotive E/E Architectures

In this thesis, we briefly introduce how electric/electronic (E/E) architectures and software architectures of vehicles look like typically nowadays.

A lot of change happened in the automotive industry, since Carl Benz drove the first vehicle in 1886 [45]. In a classical high-end vehicle available nowadays, there exist more than 70 electronic control units (ECUs), connected by more than 5 different network communication systems [155] [67] [131] (such as Controller Area Network (CAN) [179], Local Interconnect Network (LIN) [180], FlexRay [159], Media Oriented Systems Transport (MOST) [241], Peripheral Sensor Interface (PSI5), Distributed Systems Interface (DSI), Automotive Ethernet) with different transport protocols and different wiring topologies (point-to-point, line-bus (e.g. CAN, LIN), star (e.g. FlexRay) or ring (MOST)) [366]. The single communication subsystems are most often connected by a central gateway. A lot of effort is spent in the verification of correct end-to-end timing behavior of data that flows through such networks [337].

The hardware of the ECUs is mostly very heterogeneous and tailored to the specific needs of the functional feature that shall be provided by the ECU. At the time of 2005–2010, already more than two-thousand individual functions existed in some types of vehicles [67], performed by software on the ECUs. At the same time, up to 40% of the development and production costs of a car are determined by electronics and software [67] [131] [78]. Round about 90% of all innovations are driven by electronics and software [131] [155], while 50-70% of the development costs for an ECU are related to software [131]. The amount of software in a car has further increased meanwhile.

One approach to make application development independent from the heterogeneous hardware is to introduce a hardware abstraction layer and a runtime environment (RTE), like it is specified by the AUTomotive Open System ARchitecture (AUTOSAR) standardization consortium (see section 2.4.2).

However, still the number and heterogeneity of the ECUs, buses and software functions denote a huge complexity, which is still increasing [11] and therefore becomes more and more difficult to handle. Several proposals have been presented suggesting how future E/E architectures should look like to reduce the complexity back to an adequate level. Related challenges are listed for instance in [67] and [78].

One approach to reduce the complexity is the RACE approach [19] [318] [36], which is based on a study done with 240 interviews of world-wide experts from original equipment manufacturers (OEMs), suppliers, as well as political and consumer organizations [47]. The concepts of this approach are introduced in more detail in section 2.5, as it offers some basic properties of system which can be analyzed with the approach introduced in this thesis. These properties are the presence of appropriate error detection mechanisms, as well as redundancy and failover mechanisms.

Other approaches exist which tackle to better handle the existing complexity, like [6] [5], which is based on *organic computing* (OC) techniques. We do not follow OC approaches in this thesis, but give a brief overview and a discussion why we do not use them in section 3.1.6.

2.4.2 AUTOSAR

The AUTOSAR standard is developed by a consortium of nearly all automotive car manufacturers, equipment suppliers, tool suppliers and other companies and institutes associated to the automotive domain.[10] It is the successor of the OSEK/VDX consortium.[11]

The main intention of AUTOSAR is to standardize a common basis in how automotive software and system architectures should be designed and developed, to avoid that the concepts of the different

[10] http://www.autosar.org/partners/current-partners
[11] http://www.osek-vdx.org

car manufacturers diverge too much from each other and suppliers cannot anymore efficiently develop equipments for different manufacturers.

AUTOSAR is also used to integrate a high amount of functionality on one electronic control unit (ECU) in a controlled way. If mixed critical functionality is present, there are often still dedicated microcontrollers for safety-related applications [131]. However, AUTOSAR serves to reduce the number of ECUs. For instance, four non AUTOSAR ECUs are replaced by two AUTOSAR ECUs in a scenario at BMW [131]. The migration to AUTOSAR in series production is done step-by-step and also changes the working and business models, as suppliers can develop software which is applicable to multiple OEMs and the OEMs theirself concentrate on the own development of key-functions. In nowadays vehicles, normally only a subset of the electronic control units (ECUs) is developed according to AUTOSAR, but not yet all ECUs.

AUTOSAR uses a three layer concept to make application development independent from the hardware (Fig. 2.8). The lowest layer is the *basic software* (BSW) layer, containing hardware specific parts like drivers or also a standardized operating system. The middle layer is the *runtime environment* (RTE). The upper layer contains the AUTOSAR application software components, realizing the functional features used by the costumers, as well as software components to handle attached sensors and actuators. This application layer becomes hardware independent through the RTE and the BSW layer.

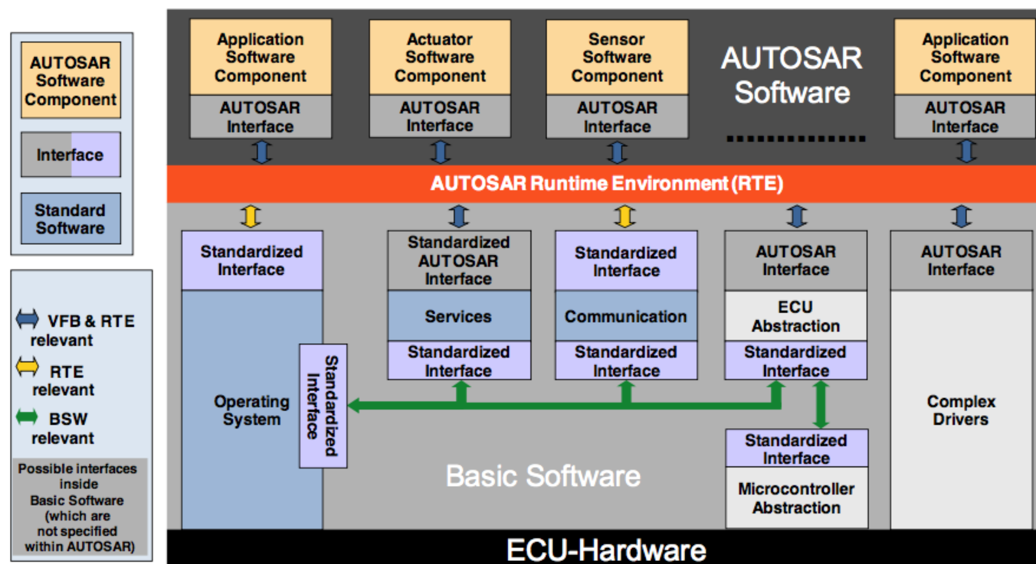


Figure 2.8: AUTOSAR layered software architecture(Fig. 3.12 in `AUTOSAR_EXP_VFB.pdf` [22])

The RTE provides a so called *virtual function bus* (VFB) to abstract from the real physical communication technology (Fig. 2.9).

In classic AUTOSAR, the RTE code is fixedly generated according to the set of application components and their interfaces, that are deployed to a certain ECU. No changes are applied at runtime to the RTE configuration. However, mechanisms exist to switch between different modes, or to stop and start (a priori known) software components. However, the standardization of a future adaptive AUTOSAR is on its way [132], planed to be released in 2017 and relaxing the fixed RTE rigidity by allowing adaptations at runtime.

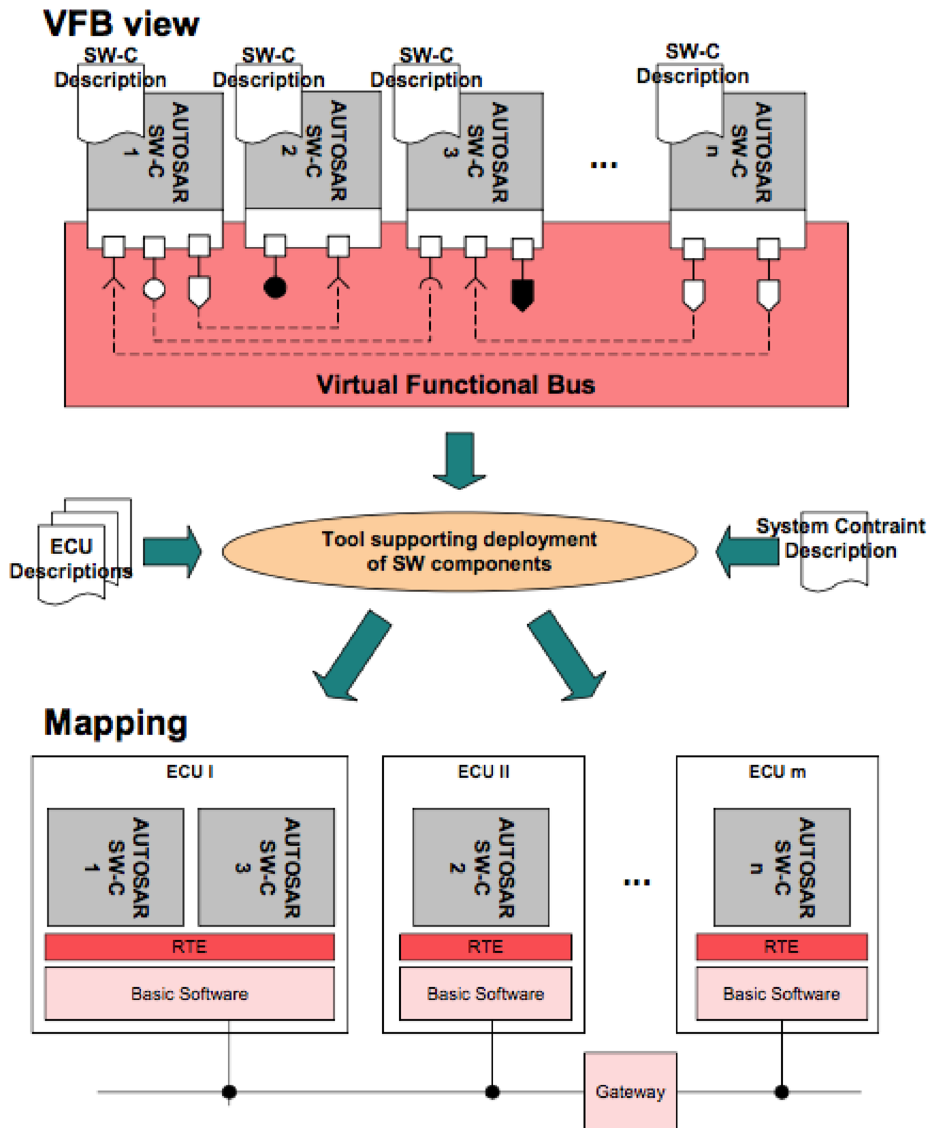


Figure 2.9: AUTOSAR Virtual Function Bus and SW deployment tool
(Fig. 2 in AUTOSAR_TechnicalOverview.pdf [20])

Fig. 2.9 also sketches the deployment of the SWCs at the application layer to the different ECUs, which then execute the SWCs. In current versions of the AUTOSAR standard [22], Fig. 2.9 is updated and more generally mentions a "Tool supporting *development* of SW components"[12], rather than a "Tool supporting *deployment* of SW components", as the definition of the deployment of SW to HW is just one activity during the software and system development.

[12]see AUTOSAR R4.2 [22], http://www.autosar.org/fileadmin/files/releases/4-2/main/auxiliary/AUTOSAR_EXP_VFB.pdf, page 10, Fig. 2.2

In this thesis, we focus on analyzing possible degradations of vehicles. Below, we show the main requirements of AUTOSAR with respect to degradation.

Requirements of AUTOSAR w.r.t. degradation of systems: The AUTOSAR standard [22] mentions the following requirements with respect to degradation:

- *AUTOSAR shall support different standardized methods to degrade the functionality of an AUTOSAR system*.[13] Depending on specific states of an ECU or of a complete system, either the full functionality cannot be available any more (example: hardware problem) or need not be available any more (example: parked car). AUTOSAR must support system and ECU degradation to properly react to such states. The main reason is to save energy. The use cases are the AUTOSAR features of *Partial Networking* and *ECU Degradation* in conjunction with *Pretended Networking* [225].

- *AUTOSAR diagnostic shall allow runtime degradation of faulty functionality to maintain minimum ECU/vehicle operability*.[14] The rationale is to maintain minimum ECU/vehicle operability in case of defect sensor values that inhibit normal performance characteristics but still allows for backup operation. An example use case is a *limp home mode* of the vehicle, in which only rudimentary driving is supported to reach home or next service station.

Mode Management in AUTOSAR AUTOSAR has a concept for mode management on software component level, ECU level and system level. In relation to this, AUTOSAR distinguishes BSW Modes (basic software, ECU level), Application Modes (on software component level) and Vehicle Modes (on system level). All these three kinds of modes can influence each other.

With respect to the analysis of degradation scenarios introduced in this thesis, the AUTOSAR mode management is one use case for applying degradation, as AUTOSAR mentions *Degradation of application functionality in certain power modes* as the use case for the requirement that *AUTOSAR shall standardize methods to organize mode management on SWC, ECU and system level*.[15]

Many other concepts beside AUTOSAR exist to support systems with multiple system operating modes, meaning that the system may have different defined structures and/or behaviors for different situations. In order to specify when the mode of a system has to be changed, mode transitions are defined depending on the current system context or user interaction. This requires that the system is context sensitive, meaning that it has knowledge about the environment and overall situation, in which it is used.

Concepts to design modes in embedded systems are also supported in MARTE [255] and AADL [295] [48], [58], [184]. Furthermore, mode based automotive systems are also tackled in [335], [334] and [240], as well as with focus on requirements engineering in [343], [342] and [340].

[13] [RS_BRF_01184] in AUTOSAR_RS_Features.pdf [22], section 4.1.1, page 26
[14] [RS_BRF_02216] in AUTOSAR_RS_Features.pdf [22], section 4.13.10, page 78
[15] [RS_Main_00460] in AUTOSAR_RS_Main.pdf [22], section 4.2.7, page 13

2.4.3 ISO 26262

The ISO 26262 [174] provides a standard to ensure functional safety in automotive E/E systems. System safety is achieved through a number of safety measures, which are implemented in a variety of technologies and applied at the various levels of the development process.

As already sketched in section 2.1.5, the ISO 26262 [174] defines different *Automotive Safety Integrity Levels* (ASIL) from QM (Quality Management) over ASIL A, ASIL B, ASIL C to ASIL D, where ASIL D denotes the highest criticality. The ASIL is determined for each hazardous event that may occur. This is done based on the *severity*, the *exposure* and the *controllability* of the considered hazardous event [174]. Finally, also the hardware and software components that are related to this hazardous event get assigned the ASIL. The higher the ASIL, the more effort has to be put into the development of hardware and software components to ensure that the components to not contain any faults that may cause failures leading to the considered hazardous event. The level QM is for non-critical components.

In the context of this thesis, we focus on the specifications of ISO 26262 with respect to graceful degradation. We introduced the definitions of graceful degradation in section 2.1.4. ISO 26262 lists graceful degradation as one of four mechanisms for error handling at the software architectural level, beside static recovery mechanisms, independent parallel redundancy and correcting codes for data. These four mechanisms are evaluated w.r.t. to functions with different ASIL and recommendations are given about which mechanisms should be used for which ASIL. The usage of graceful degradation is recommended (+) for ASIL A and B and highly recommended (++) for ASIL C and D, which is the overall highest recommendation of all four mechanisms (see Fig. 2.10).

Methods		ASIL			
		A	B	C	D
1a	Static recovery mechanism[a]	+	+	+	+
1b	Graceful degradation[b]	+	+	++	++
1c	Independent parallel redundancy[c]	o	o	+	++
1d	Correcting codes for data	+	+	+	+

[a] Static recovery mechanisms can include the use of recovery blocks, backward recovery, forward recovery and recovery through repetition.

[b] Graceful degradation at the software level refers to prioritizing functions to minimize the adverse effects of potential failures on functional safety.

[c] Independent parallel redundancy can be realized as dissimilar software in each parallel path.

Figure 2.10: Graceful degradation and other mechanisms for error handling at the software architectural level, from [176] (Table 5)

The second mechanism, beside *graceful degradation*, which is shown in Fig. 2.10 and which we consider in this thesis, is *independent parallel redundancy*. This means not only to provide redundant backups of a software component, but to establish diversity by providing a second implementation of the same software component, which is dissimilar. We consider a special form of this, namely diversity combined with degradation, in context of our analysis approach in section 4.7.

2.5 The RACE Approach

In the Project RACE [16], a new fault-tolerant software and system architecture for future electric vehicles and comparable systems was investigated and implemented in two demonstrator cars.

The motivation is to reduce the complexity of the electric/electronic (E/E) architecture of vehicles in order to enable a more cost efficient and safe development of innovative vehicle features, like autonomous driving, that are partially required to behave fail-operational. Another motivation is also to tackle development goals that address three global megatrends: 1) zero emissions to address climate change, 2) intelligent mobility to address urbanization, and 3) zero accidents to address demographic change [118].

The main idea is to use a centralized computation approach, as proposed in [47], to overcome the very complex highly distributed and highly heterogeneous state of the art E/E architectures as described in section 2.4.1. A runtime environment (RTE) is introduced that offers generic fault-tolerance mechanisms [36] and incorporates failover and degradation strategies.

As introduced in section 2.4.1, the electric/electronic (E/E) architectures of nowadays state of the art vehicles has been historically grown over the last four decades and are heterogeneous mixtures of round about 70 electronic control units (ECUs) and 5 different bus systems [131]. Also the functional software complexity has been grown drastically by introducing more and more software based functional features, which are more and more interacting. This complexity requires high effort and cost to integrate additional features and to ensure high quality by testing or other quality assurance methods.

The AUTOSAR approach already tackles a lot of these problems by introducing a Runtime Environment Layer (RTE) to abstract from the lower hardware layer and providing a so called Virtual Function Bus (VFB) to abstract the physical communication paths from the application layer, which is executed on top of the RTE. However, AUTOSAR does not tackle the reduction of the systems E/E hardware complexity directly. It tries to mitigate the negative effects of current E/E architectures to the application software.

The RACE platform architecture extends the idea of AUTOSAR with a blueprint system architecture and generic mechanisms to enable functional features with fail-operational behavior. Instead of tailoring the RTE code for the set of application software components running on a certain ECU - like it is done in AUTOSAR - the RACE RTE is fully configurable. The configurability supports changes of the set of deployed application software components. This introduces a certain runtime overhead, but is a key for changing the system in field, like towards Plug&Play extensions of vehicles with new software-based functional features and new physical sensors and actuators.

The main concepts of the RACE platform are introduced subsequently in the following sections, providing basic knowledge for the fault-tolerance concepts and assumed system properties being considered later when introducing our analysis approach in section 4.

2.5.1 Software and System Architecture

Platform Layout: The RACE platform consists of a scalable set of centralized computation units, called the *Central Platform Computer (CPC)*. Attached to the CPC is a set of so called *Smart Aggregates* for physical sensing and actuation. The smart aggregates are highly integrated and execute local control loop algorithms, while the CPC executes higher level control strategy applications. The physical communication is established by a redundant reliable network, based on switched Ethernet [19]. A sketch of such a platform architecture is shown in Fig. 2.11.

The middle box of Fig. 2.11 shows an exemplary CPC with four so called *Duplex Control Computers (DCCs)* as central computing units. A DCC builds a Dual-Modular Redundancy (DMR) control unit, where each operation is calculated on two parallel execution lanes L1 and L2 and the outputs are compared

[16] Robust and Reliant Automotive Computing Environment for Future eCars; funded by the German Federal Ministry for Economic Affairs and Energy (BMWi) under grant no. 01ME12009, carried out from early 2012 until early 2015, `http://www.projekt-race.de/en`

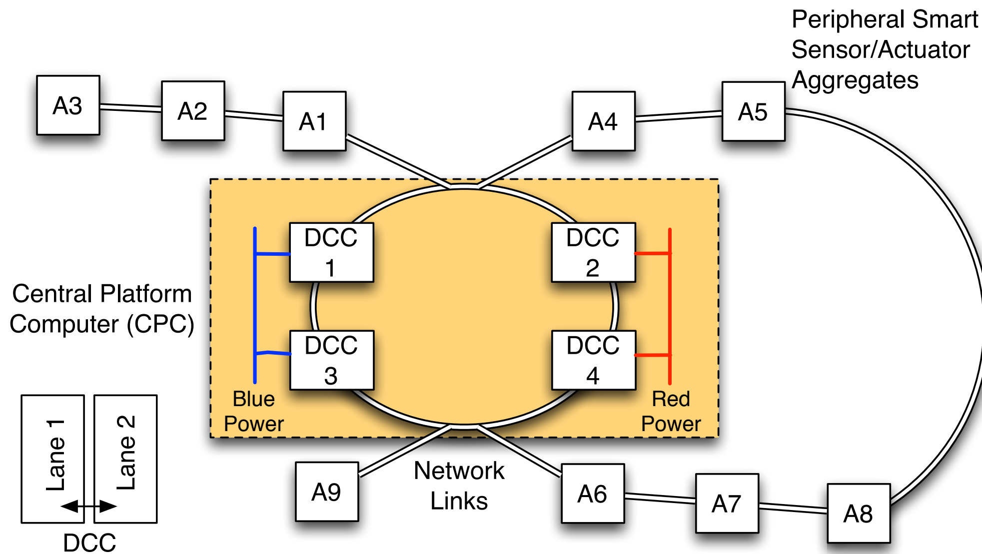


Figure 2.11: Example instance of the RACE hardware architecture [35]

for bit-identicality. Software components that realize fail-operational features are deployed to at least two DCCs, building a Duo-Duplex Redundancy (DDR). See section 2.1.6 for the definitions of DMR and DDR, as well as section 2.5.2 for a deeper introduction into the RACE redundancy concept.

Attached to this CPC are in this example nine smart aggregates, five of them in a ring topology (right) and the other four divided on two further communication branches (left). Each aggregate can be accessed from each DCC over the reliable switched ethernet.

Runtime Environment (RTE): The RACE Runtime Environment (RTE) is a modular middleware layer between the operating system of the execution units and the application software components (ASWCs), see Fig. 2.12. It is for instance responsible for enabling data communication between the different entities of the system, like application software components and sensor/actuator aggregates. To enable this, it is executed on all hardware execution units (DCCs and smart aggregates).

The RTE is designed in a data-centric manner, ensuring that from each DCC all sensor data can be accessed and all actuators can be controlled [318]. This enables that application software components (ASWCs) can deliver their service on all DCCs. This means that it is irrelevant which ASWC is located on which DCC, offering flexibility in the decision about the deployment of ASWCs to the DCCs. Such flexibility is a major goal in software design for vehicles [230].

Also data-fusion and data-distribution features are offered by software components of the RTE. For instance, data-fusion mechanisms allow to receive data from sensor aggregates, which are present with different levels of redundancy [36]. The redundantly sensed data is merged by the RTE to a single sensor value, which is then delivered to the ASWCs that need this sensor value. More about redundancy in RACE will be presented in section 2.5.2.

The definition of data communication dependencies is realized using the publish/subscribe communication paradigm [116]. Each ASWC and each aggregate defines the amount of required input data by subscriptions, and the amount of provided output data by publications. More about this will be introduced later in section 2.5.3.

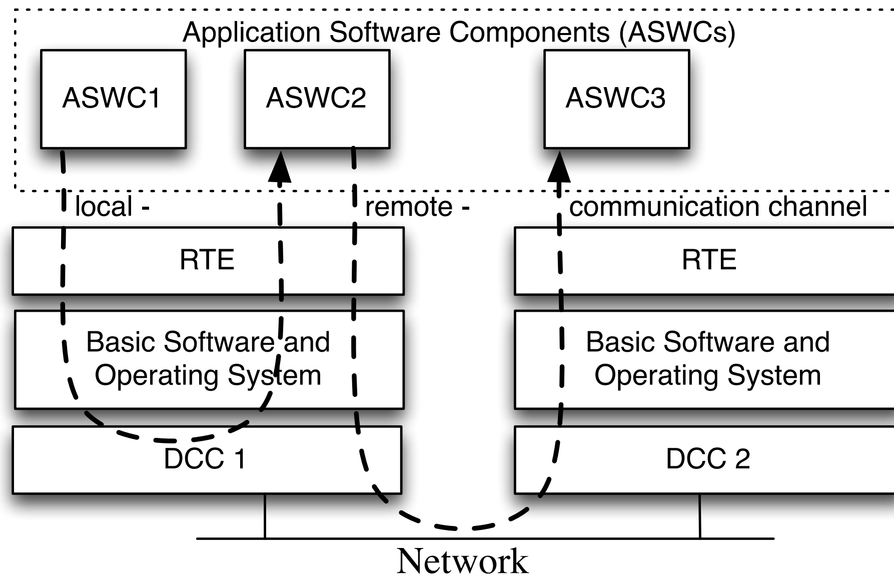


Figure 2.12: RACE layered Architecture with RTE

To provide fault-tolerance, the RACE RTE contains inherent mechanisms for error-detection, consolidation and error-handling [129]. Detection is done by monitoring all communicated values w.r.t. plausibility of values, like minimum/maximum range of valid values or maximum delta change between two consecutive values. Also the timing of communication is monitored, like network frames that are received too late. Mechanisms exist to isolate faulty hardware and software components from the remaining system to avoid harm. For this, different fault-containment regions (FCRs) are defined. Also a failover mechanism is provided to support fail-operational features. More about this is introduced in section 2.5.2. A more detailed introduction into the principles of the RACE RTE and an overview over the different RTE components is published in [318], [129] and [36].

Scheduling: The RACE platform operates in fixed time-triggered real-time execution cycles, using the concept of *Logical Execution Time* (LET) [200]. This abstracts from the physical execution time on a particular device and thereby abstracts from both the underlying execution hardware and the communication topology. The LET forms a basis for component-oriented development of real-time systems. Languages such as Giotto [161] harness LET abstractions.

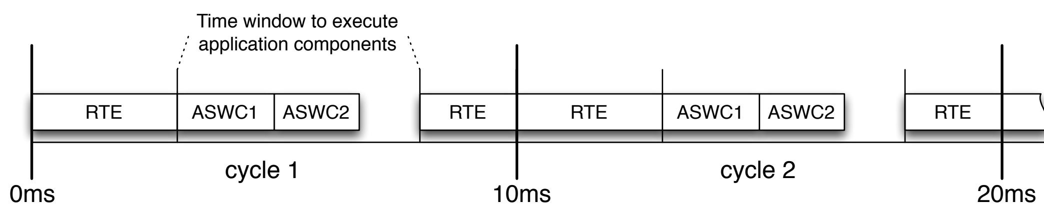


Figure 2.13: Cyclic execution of RTE and ASWCs

The RACE RTE executes the application software components (ASWCs) in fixed execution cycles. An example is shown in Fig. 2.13, in which two ASWCs are executed within an execution cycle of 10ms.

Also the different RTE components (e.g., for network data transmission, data validity check, data fusion, error handling, etc.) are executed within the same cycle. Hence, only a subset of the cycle time is available for executing ASWCs. The RTE transfers the communicated data at each cycle border. The input data for all ASWCs is read in from the network at the beginning of each cycle, and the output data of the ASWCs is collected and distributed to the subscribers locally and over the network at the end of each cycle. This means that any change of input data taking place during a cycle only affects the next cycle, not the current cycle. Hence, precedence relations between ASWCs have not to be considered in the schedule of the ASWCs, as the input data that is given to an ASWC is always the data that is produced in the previous cycle, not in the current cycle. The system guarantees that data that is sent in cycle x is available at each execution unit (DCCs and smart aggregates) at cycle x+1. Of course, an appropriate network communication mechanism is required for this, introduced briefly below.

Networking: The RACE platform uses an Ethernet based network communication topology. A scalable ring-based full-duplex switched Ethernet architecture is used, where an inner ring connects the DCCs of the CPC (cf. Fig 2.11), and additional outer rings or branches connect the aggregates to the CPC. Logical communication channels are automatically established by the RTE, based on the publish/subscribe descriptions of provided and required data items of ASWCs. Based on these logical communication channels, the contents of the physical network frames are configured, that have to be transmitted between the DCCs and aggregates. In each execution cycle, a DCC receives network frames from other DCCs and aggregates, executes the deployed ASWCs and transmits the output data to other DCCs or aggregates. The network distinguishes between critical frames, containing critical data, and non-critical frames, containing non-critical data. The network guarantees that critical network frames, which are sent in cycle x, are available at all other execution units at cycle x+1, before the time at which network frames are read in during cycle x+1. To be able to guarantee this, the maximum size of critical network frames is limited and frame preemption on network transmission level is used to reduce the network transport delay for critical frames by allowing them to preempt the transmission of non-critical frames [36]. This mechanism is currently being standardized in IEEE802.1Qbu (Time-Sensitive Networking, TSN) [173]. Automotive BroadR-Reach single-pair cabling is applied. Further details about the RACE network communication are provided in [19] and [318].

2.5.2 Safety and Fault-Tolerance Concept

In this section, we will briefly introduce those concepts of the RACE approach for ensuring safety and fault-tolerance, that build the basis to understand the requirements for the formal system model and formal constraints, introduced later in this thesis in chapter 4.

ASWC-Clusters: ASWCs with identical criticalities (e.g. in form of identical ASIL and fail-operational requirements) are grouped into so called *ASWC-Clusters*. The motivation for these clusters is to reduce the complexity of mechanisms for the separation of mixed critical components and of mechanisms for runtime error handling. Hence, the clusters are just a management unit. However, ASWCs with identical criticalities can also be grouped into different ASWC-Clusters, if desired. But each ASWC is mapped to exactly one cluster.

Separation: ASWCs with different criticalities have to be separated to avoid bad influence from low critical to high critical ASWCs. This separation has to be established in form of *spatial* and *temporal separation* of mixed critical ASWCs. In RACE, this is reached by running the RTE on top of the PikeOS[17] operating system and using partitioning mechanisms of PikeOS.

[17] `http://www.sysgo.com/products/pikeos-rtos-and-virtualization-concept`, last access at 30 November 2015

Fault-Containment-Regions (FCR): Different *fault-containment regions* (FCR) exist in different granularity [128]. The FCRs deal as isolation units in order to avoid propagation of errors between FCRs. Beside others, examples for FCRs are DCCs, aggregates and ASWCs.

Runtime Error Detection and Error Handling: The RACE RTE offers error detection and error handling mechanisms for mixed critical components. All input data is checked for validity, before it is forwarded to the ASWCs. If a value is detected as invalid, the RTE substitutes it with the last valid value or a default value, together with an attribute which informs the ASWC about this. The choice of the kind of substitution depends also on how long a value is already invalid. Also the timing of network frames and the execution of ASWCs in their worst case execution time (WCET) boundary is checked.

At runtime, an algorithm obtains the so called *performance levels* of all DCCs, aggregates, ASWCs and ASWC-Clusters and triggers isolations, failovers or degradations, if necessary [18]. In case a DCC becomes isolated, some of the ASWC-Clusters that were used to be executed on this isolated DCC may have to be resumed on other intact DCCs. These are those ASWC-Clusters that contain ASWCs with fail-operational requirements. The RTE manages this failover switching at runtime. The failover switching after isolations of DCCs is done on cluster level, while the isolation of single failed ASWCs is handled on ASWC level. Hence, if an ASWC inside a cluster fails and becomes isolated, the other ASWCs within this cluster can continue their operation and because ASWCs are fault-containment regions (FCRs), the other ASWCs will not be negatively affected. Further information about the detection and handling of errors of input data in RACE are provided in [128], [126] and [36]. Additional approaches for error detection and diagnosis are for instance described in [181].

Non-intrusive fault-injection tests: The RACE approach supports non-intrusive data monitoring and non-intrusive fault-injection [124] [36] [125], offering the manipulation of data or state information within the system during runtime and by this, allowing to test if the systems fault-tolerance mechanisms work properly as specified.

Redundancy Concept: RACE uses a variety of redundancy mechanisms to enable fail-operational features. On hardware level, there are redundant sensors, redundant actuators, redundant physical communication links, as well as redundant aggregates and redundant central execution units (DCCs). Different degrees of redundancy exist on sensor/actuator level, on aggregate level and on communication link level. For instance, the Steering-Wheel in the Steer-By-Wire demonstrator vehicle is sensed by two redundant aggregates, each having three redundant sensors [36]. Hence, overall six sensor-values are captured. Each aggregate sends the sensed values over two redundant communication links towards the CPC. Hence, overall twelve values are received at the DCC, distributed on four network packages, each containing three values. The RTE at the receiving DCC checks these values for plausibility and integrity and finally fuses the values to one single value that is given to the ASWCs that subscribe this value. However, this form of sensor and actuator redundancy is not further handled in the context of this thesis.

Each central DCC has a Dual-Modular Redundancy (DMR), as they comprise two computation lanes, executing the same instructions in parallel. A pair of two DCCs builds a Duo-Duplex Redundancy (DDR). More generic, multiple DCCs build a N-Duplex Redundancy. Due to this, a fail-operational feature can be kept active during at least $N-1$ hardware failures of DCCs, if the ASWCs that realize this feature are redundantly deployed onto N DCCs.

The RACE runtime environment (RTE) checks that the both execution lanes of a DCC provide bit-identical results. In case of a hardware failure in one execution lane, the results may become unequal, what is detected by the RTE and results in an isolation of the DCC. These DCC isolations are the major scenario considered in this thesis.

On software level, redundancy is applied in form of a dynamic redundancy mechanism. For each ASWC, one active master instance is deployed. Additional hot- or cold-standby slaves (also called hot/cold

spares) are deployed redundantly for those ASWCs that realize features which are required to behave fail-operational. A fail-operational ASWC that was used to be executed on an isolated DCC is resumed on another DCC by enabling a hot- or cold-standby slave to become the new master. A hot-standby slave is a passive replica, as its outputs are ignored, but its internal state is updated and kept synchronous to the state of the active master. The cold-standby slaves are not executed, but just exist in the memory of DCCs, ready to be started if required. However, they start from their initial state. Also combinations of hot- and cold-standby slaves exist if an ASWC is required to survive more than one DCC isolation. For instance, if an ASWC is required to survive three DCC isolations (we call it to have a fail-operational level of three), three slaves are required in addition to the master. One of these three slaves will potentially be a hot-standby slave, but the other two slaves will initially be cold-standby slaves. In case the master is lost and the hot-standby slave becomes the new master, then one of the both cold-standby slaves becomes the new hot-standby slave.

Minimum fault-tolerance time (minFTT): Each application has multiple safety goals, while each safety goal has an assigned *fault-tolerance time* (FTT), defining how long an interruption of the provision of that application can be tolerated without risk. The smallest of these FTTs is the *minimum fault-tolerance time* (minFTT) of an application. The minFTT is also specified for the application software components (ASWCs), which implement an application [35]. In this thesis, we use the term *functional feature* instead of the RACE term of application.

Fault recovery time (FRT): The RACE system ensures a maximum time to perform a *failover*. This means, in case a master ASWC instance is lost due to an isolation of a DCC or a failure of the master ASWC itself, the time that is required to activate a corresponding cold-standby slave instance of that ASWC to let it become the new master. This failover time is called the *fault recovery time* (FRT) of the system. The FRT can be defined as a constant for the whole system, as it can be shown that a maximum FRT can be proven for the RACE concept (not part of this thesis). During the project, the FRT was aimed to be at most 50ms [35].

As a standby slave is required for a fail-operational ASWC to provide redundancy, not always a hot-standby slave is required. Sometimes, a cold-standby slave is sufficient. The decision, if a hot-standby slave is required or if a cold-standby slave is sufficient, depends on the minFTT of the considered ASWC compared to the *fault-recovery time* (FRT) of the system. If the minFTT of the ASWC is smaller than the system FRT or equal to it, then a hot-standby slave has to be established, otherwise a cold-standby slave is enough. This means, if the failover mechanism to activate a cold-standby slave is quick enough to beat the minFTT of this ASWC, then a cold-standby slave is sufficient to resume the ASWC in time. Otherwise, if the failover mechanism to activate a cold-standby slave is not quick enough, we need a hot-standby slave, as this can become a master much faster.

Normal-Law and Direct-Law Components: The RACE concept offers to use so called Normal-Law (NL) and Direct-Law (DL) software components, providing basically a functionality of similar kind, but with different functional quality. The intention behind this is that the NL component provides the normal full-fledged feature and the DL component provides a degraded backup with basic functionality. If the NL component cannot be provided anymore, for instance due to a faulty required sensor value or due to insufficient other resources, then the simpler DL component can be activated, assuming that this requires less resources, which are available sufficiently. The availability of the DL component mitigates the loss of the NL component. As ASWCs realize functional features, this denotes a degradation mechanism on feature level. A simple example is assisted driving in NL and basic manual driving without assistance in DL. In this thesis in section 4.7, we introduce and discuss some assumed design principles about the realization of features with NL and DL ASWCs, and enable our analysis approach to analyze architectures that follow those assumptions.

Further details about the RACE safety and fault-tolerance concept are published in [128], [127], [129], and [291]. The concepts are partially based on ideas that have been introduced in [17] and [281] in context of the research project SPARC (Secure Propulsion using Advanced Redundant Control).[18]

2.5.3 Application Development and RTE Configuration

Designing RACE Applications: RACE application software components (ASWCs) as well as physical aggregates are delivered with self-describing information contained in so called *manifests* [318]. A manifest contains all data that is required to integrate a software component or an aggregate into a vehicle design [36]. For instance, manifests contain component interfaces in form of publications and subscriptions of topics and attached attributes, as well as the *Worst Case Execution Time (WCET)* of the cyclic executable function and safety relevant information such as the components requirement to behave fail-operational. The set of possible topics and attributes in one system is predefined in a so called *dictionary* [71]. Topics may be for instance physical properties (temperature, pressure, etc.) or system data (e.g. recognized objects in front of the vehicle, a trajectory to drive, etc.). Attributes describe the instances of a topic, such as the location and meaning of a temperature and the unit of measurement. The dictionary concept enables data compatibility between applications of different suppliers, as they build their applications on top of this common topic dictionary.

A RACE specific extension of the CHROMOSOME Modeling Tool (XMT) [71] is used to describe the manifests and the dictionary. The XMT tool also offers to analyze virtual compositions of manifests in integrated product models. During the analysis, for instance the logical data-flow between the software components and aggregates is checked for completeness and unambiguity [308].

Finally, manifest files are generated together with code wrappers of the software components. In the wrappers, all read/write operations for the modeled publish/subscribe interfaces are already present, as well as an initialization function and a function which is executed by the RTE in each cycle. Into this template, the developer can integrate the application behavior, for instance code generated from Matlab/Simulink.

Due to the inherent safety mechanisms provided by the RTE, application development is facilitated as developers can focus on implementing the functional features and have not to check plausibility of input data themselves. Instead, the plausibility of values is checked by the RTE.

RTE Configuration The RTE configuration is based on the set of given manifests. The configuration may happen at system design time, or at system start-up, or later during a Plug&Play scenario. A configuration component of the RTE collects all the information from the given manifests and executes plausibility checks to decide about the composability of the manifest set [36].

For instance, this includes checking that the sum of the WCETs of the actively deployed ASWCs does not exceed the given time-budget in the execution cycle on each execution unit (cf. scheduling in section 2.5.1). Additionally, a graph is calculated containing the logical data-flow between the ASWCs and the aggregates, based on the publications and subscriptions that are defined in the manifests [71]. Based on this graph, the data-flow is checked for ambiguity and completeness, as well as where data-fusion is required with which properties. If for instance multiple publishers send data to the same subscriber in a scenario where data-fusion is not intended, this is detected as a problem. Also safety relevant properties are checked, like the level of redundancy that certain ASWCs require. In case of conflicts or other problems, the configuration process is aborted and the integration is rejected. After a successful analysis, finally the configuration data structures for the RTE components are created and the RTE is put into operation.

[18] Funded by EU FP6, 2004-2007, `http://www.transport-research.info/project/secure-propulsion-using-advanced-redundant-control`, last access at 30 November 2015

2.5.4 Demonstrator Vehicles

Two demonstrator vehicles were developed within the RACE project. The first car shows an evolutionary migration path from current E/E architectures towards the RACE architecture, by replacing some parts of the former E/E architecture by RACE technology. [19] The second car was a revolutionary new car completely based on the RACE technology [72]. This revolutionary car contained a steer-by-wire application, which is highly safety critical and therefore realized with redundant steering wheel and steering rack aggregates [36].

2.6 Avionic Architectures and Standards

Beside the Automotive domain, the Avionic domain might be a second domain in which the approach presented in this thesis can be applicable.

Foundations in Avionic Architectures and Standards: In the avionic domain, a fault-tolerant modular software and system platform architecture has been standardized with *Integrated Modular Avionics* (IMA) [353] [260], defined in DO-297. [20] A standardized software interface for IMA is given by ARINC 653 [260] [277]. [21] The avionic specific standard DO-178C [22] describes how to develop software for avionic systems safely. Supplement specifications exist, describing for instance how to apply formal methods (DO-333[23]) or how to apply model-based development and verification (DO-331[24]).

To separate mixed critical system elements, spatial and temporal partitioning mechanisms have been developed, like introduced in [292] and [356] and specified in ARINC 653 [260] [277].

Many parts of avionic systems have high demands to behave fail-operational [157]. Hence, redundancy is applied where necessary. For instance, the fly-by-wire subsystem is designed redundantly [338], in fact in the Boing 777 in a triple-triple modular redundant manner including dissimilarity [358] [359].

Degradation Concepts in the Avionics Domain: In the avionic domain, *control law* concepts have been developed, including degradation concepts. For instance, Airbus airplanes have multiple so called *Flight Control Laws*, present for different flight operating modes, like *Ground Mode* or *Flight Mode*, applied in different flight situations. The control laws are Normal law, Alternate law, Direct law, and Mechanical law. [25, 26]

In the *Normal law*, a lot of protection mechanisms are active, like Attitude Protection. These software based protection mechanisms prevent a lot of dangerous flight situations. In case of failures of system entities, the airplane can switch into different levels of the *Alternate law*. In these Alternate laws, some of the protection mechanisms are lost, but the plane is still under control. If even more or worse failures appear, the airplane can be switched into *Direct law*, providing a direct relationship between sidestick and control surface, without any protection mechanisms. For the very worst case scenario, after a complete loss

[19] http://www.projekt-race.de/en/news/archive/siemens-to-equip-streetscooter-electric-vehicle-with-innovative-electronics-and-software.php, last access at 30 November 2015

[20] http://www.rtca.org/store_product.asp?prodid=617, DO-297 Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations, 2005

[21] http://store.aviation-ia.com/cf/store/catalog_detail.cfm?item_id=496, http://store.aviation-ia.com/cf/store/catalog_detail.cfm?item_id=2495, ARINC Specification 653: Avionics Application Software Standard Interface

[22] http://www.rtca.org/store_product.asp?prodid=803, DO-178C Software Considerations in Airborne Systems and Equipment Certification

[23] http://www.rtca.org/store_product.asp?prodid=859, DO-333 Formal Methods Supplement to DO-178C and DO-278A

[24] http://www.rtca.org/store_product.asp?prodid=815, DO-331 Model-Based Development and Verification Supplement to DO-178C and DO-278A

[25] https://en.wikipedia.org/wiki/Flight_control_modes, accessed 22th Sept. 2015

[26] http://www.airbusdriver.net/airbus_fltlaws.htm, accessed 22th Sept. 2015

of electrical flight control signals, even a *Mechanical law* is present, providing pure mechanical back-up control possibilities for pitch and lateral control, supported by hydraulic power.

2.7 Foundations in Satisfiability Solving and Optimization

Fundamentally, many combinatorial satisfiability and optimization problems can be formulated as a *Constraint Satisfaction Problem* (CSP) [214]. In a CSP, *hard constraints* can be expressed that must be satisfied, as well as *soft constraints* that should be satisfied, but may be violated. A *weight* is added is added according to the degree of preference for each soft constraint, the problem becomes an optimization problem [62]. The problem is to find a variable assignment to all variables that satisfies all hard constraints and at the same time optimizes a global cost function for the soft constraints. Such an optimization problems is sometimes also called *Constraint optimization problem* (COP) [266]. For instance, CSP is applied to solve an allocation problem in [167].

Below we give a brief overview over different technologies in problem satisfiability solving and problem optimization. They can be seen as forms of CSP respectively COP.

Linear Programming: Coming from classical *Operations Research* (OR) domain, creating quantitative models to support decisions for advanced optimization problems.

- LP (linear programming) (e. g., solved by *Simplex* [151]). Special cases exist, like *stochastic linear programming* [191]
- ILP (integer LP) (e. g., Branch and Bound, applied in [262])
- MILP (mixed ILP) (e. g., Gurobi [27])
- MO PB-ILP (multi-objective pseudo-boolean ILP), the ILP problem has binary variables and multiple conflicting objectives [228]

Metaheuristic Search Algorithms: A Metaheuristic is to automatically determine an appropriate heuristic to find a non optimal, but sufficiently good solution for a given search problem, also usable to heuristically solve an optimization problem. The benefit prior to an exact optimal algorithm is that the computational complexity is reduced, such the search can be performed more efficient in less time. This is especially relevant for NP-hard problems. The following bullet points lists just some metaheuristic approaches:

- Simulated Annealing (SA) [199] [329] is a probabilistic technique to approximate a global optimum in large search spaces. Inspired by the technique of annealing in metallurgy (applying slow cooling), SA algorithms slowly decrease the probability of accepting worse solutions, while exploring the solution space. Local optima can be forsaken in order to find other better optima.
- Tabu Search (TS) [137] [253] guides a local heuristic neighborhood search procedure to explore solutions beyond a local optimum, by using a so called *tabu list*. Hence, TS is able to cross boundaries of feasibility or local optimality. There are also TS approaches addressing the CSP [253].
- Evolutionary Algorithms (EA) [24], also used for multiobjective optimization [121], for instance implemented in the *Multi Objective Evolutionary Algorithms* (MOEA) Framework. [28]

[27] http://www.gurobi.com
[28] http://moeaframework.org

- Genetic Algorithms (GA) [170] (e. g., used in [9], [254], [204], [329]). Genetic algorithms belong to the larger class of evolutionary algorithms (EA).
- Swarm intelligence approaches, like
  - Particle Swarm Optimization (PSO): a concept for the optimization of continuous nonlinear functions, introduced in [194], having ties to both genetic algorithms (GA) and evolutionary programming (EP).
  - Ant Colony Optimization (ACO): introduced in [103]

In [228], a classification is done into approaches applying 1) Continuous Encoding (e. g., Particle Swarm Optimization), 2) Discrete Encoding (e. g., Simulated Annealing), and 3) Mixed Encoding (e. g., Evolutionary Algorithms).

The Metaheuristic search algorithms can be further distinguished into algorithms applying 1) *local* (neighborhood) search strategies (e. g., Tabu Search) and 2) *global* search strategies (e. g., Genetic Algorithms and Particle Swarm Optimization). The Simulated Annealing approach mediates between local and global search.

Other Generic Decision Making Approaches:

- Answer set programming (ASP) describes a problem as a logic program, a set of axioms[29] and a goal statement, under the answer set (stable model) semantics of logic programming [135] in such a way that the models of the program (answer sets) correspond to the solutions of the problem [87].
- Satisfiability (SAT) solvers for boolean formulas, checking if there exist values for the boolean variables, such that the formula evaluates to *True*.
- Satisfiability modulo theories (SMT) solvers. Satisfiability Modulo (the) Theory T — SMT(T) — is the problem of deciding the satisfiability of Boolean combinations of propositional atoms and theory atoms [83]. Examples of useful theories are equality and uninterpreted functions, difference logic and linear arithmetic (either over the reals or the integers), the theory of arrays and bit vectors, as well as combinations of those theories. Implementations are, for instance, Yices [30], MathSAT [31], Z3 [93] [50] and CVC4. [32]
- SMT optimizers. SMT solvers with optimization facilities. One implementation is the new version of Z3 [33], also called Z3Opt or νZ [51] [52]
- EF-SMT: Exists/Forall-SMT solvers extend SMT from top-level quantified forall problems to exists-forall problems [82] [106]. EF-SMT solvers use an ordinary SMT solver and iteratively perform guessing of instantiations for the Exists variables and querying the SMT solver with the resulting Forall formula. If this fails, the result (i.e., counterexample) of the Forall query is used to find the next (better) instantiation of the Exists variables, until is succeeds [293].

A more detailed introduction into many of the mentioned and some additional solving and optimization technologies is presented in [227].

In this thesis, we do not aim in developing a new efficient solving and multi-objective optimization technology tailored to the discussed problem. Instead, we use a generic off-the-shelf framework to get

[29] An *axiom* is a statement that is valid for any assigned values of the domain, like the commutative addition axiom "a+b = b+a".
[30] `http://yices.csl.sri.com`
[31] `http://mathsat.fbk.eu`
[32] `http://cvc4.cs.nyu.edu/web`
[33] `https://github.com/Z3Prover/z3`

solutions to the problems that arise in this thesis. We apply the Z3 SMT solver as problem solving and optimization technology. The reason for this choice is that it is very well documented, well supported, used in commercial context [138] but freely available under MIT license [34], has an API to various programming languages like C, Python and Java, is relatively efficient for a lot of logics compared to other SMT solvers [35] and since [52] it also supports the definition of multiple objective functions, whose optimization conflicts can be resolved in different ways. Time efficiency (performance) in calculating these solutions is not in focus of this thesis.

[34] https://github.com/Z3Prover/z3/blob/master/LICENSE.txt
[35] http://smtcomp.sourceforge.net/2015/results-summary.shtml

CHAPTER 3 Related Work

In this chapter, we discuss existing related work. On the one hand, in section 3.1 we focus on related design and analysis approaches for fault-tolerant systems, particularly on analyzing graceful degradation. On the other hand, in section 3.2 we discuss synthesis approaches for optimized system design decisions, particularly for the deployment of software components to hardware execution units.

Contents

3.1 Approaches to Design and Analyze Fault-Tolerant Systems 41
3.1.1 Design and Analysis of Graceful Degradation 42
3.1.2 Design and Analysis of Reliability and Robustness 48
3.1.3 Design and Analysis of Availability 51
3.1.4 Fault-Tolerant Scheduling for Mixed Criticality Systems 51
3.1.5 Design of Structural and Behavioral Reconfiguration 52
3.1.6 Self-x Approaches 53
3.2 Constraint Based Synthesis of Design Decisions 58

3.1 Approaches to Design and Analyze Fault-Tolerant Systems

Examples of fault-tolerant safety critical systems: A lot of research has been performed in the area of designing safety critical systems in a fault-tolerant manner. Particularly, huge effort has been spent in the avionics domain, for instance, to design dependable Fly-by-Wire aircraft [339], or to design partitioning mechanisms to ensure fault containment and avoid fault propagation between functions that share resources in the scope of *Integrated Modular Avionics* (IMA) [292]. Also in the automotive domain, system *safety* is very important [174], and system *reliability* by fail-operational fault-tolerance becomes increasingly important [317] [203], especially in the context of automated and autonomous driving features [355] [72]. Efforts to design fault-tolerant vehicles have been spent for instance in the projects DySCAS[1] [14] [79], RACE[2] [318] [36], SafeAdapt[3] [289] [263] and SAFER[4] [196]. RACE focused particularly on enabling fail-operational driving features, see also section 2.5.2. Another domain in which safety and fault-tolerance are important are *cyber-physical systems* (CPS). CPS are introduced in [66] to be *open adaptive globally connected cooperative self-organizing software-intensive embedded systems of systems* (aggregation of CPS attributes mentioned in [66]). For instance, a safety roadmap for CPS has been presented in [336], mentioning that the *openness* (dynamic connections to other systems) and *adaptivity* (adapt to changing environmental contexts) of CPS requires enhanced assurance methods to ensure safety and reliability, to cover the runtime dynamism of the overall CPS architecture.

In this thesis, we focus on the automotive domain. We now highlight and discuss work related to this thesis in different categories.

[1] Dynamically Self-Configuring Automotive Systems

[2] Robust and Reliant Automotive Computing Environment for Future eCars / Reliable Automation and Control Environment, http://www.projekt-race.de

[3] Safe Adaptive Software for Fully Electric Vehicles, http://www.safeadapt.eu

[4] System-level Architecture for Failure Evasion in Real-time Applications

3.1.1 Design and Analysis of Graceful Degradation

W. Nace, P. Koopman (RoSES Project, CMU Pittsburgh, 1999-2001): In [247] [248] [246], the authors present an approach to design gracefully degrading distributed embedded systems. The presented framework is separated into three steps. 1) First, they model a feature model containing the superset of all features that may be offered by the products of a product family. When designing a product out of the product family, those features are selected that provide the desired functionality of the product. A feature selection algorithm optimizes which features get picked to provide which functionality. 2) In the second step, software components are selected that fulfill the requirements of the features that were selected in the first step. The selection is done out of a library of components.[5] 3) In the third step, they calculate a feasible allocation (alias deployment) of the selected software components to the microcontrollers of the system. This deployment is calculated by using a *bin-packing* algorithm, but they write that this choice was quite arbitrary and many other algorithms (like integer programming) could have been chosen as well. They apply functional *redundancy* as fault-tolerance technique.

Beside the calculation of deployments with redundancy in step three, the most related part to the work presented in this thesis is a model about sequences of hardware failures and an analysis of graceful degradation in these sequences. They model hardware failure sequences as a lattice. Each vertex of the lattice represents a set of intact available hardware components (microcontrollers). It is a lattice as multiple top level vertexes exist, each representing a different product of the product family. Fig. 3.1 shows the lattice concept.

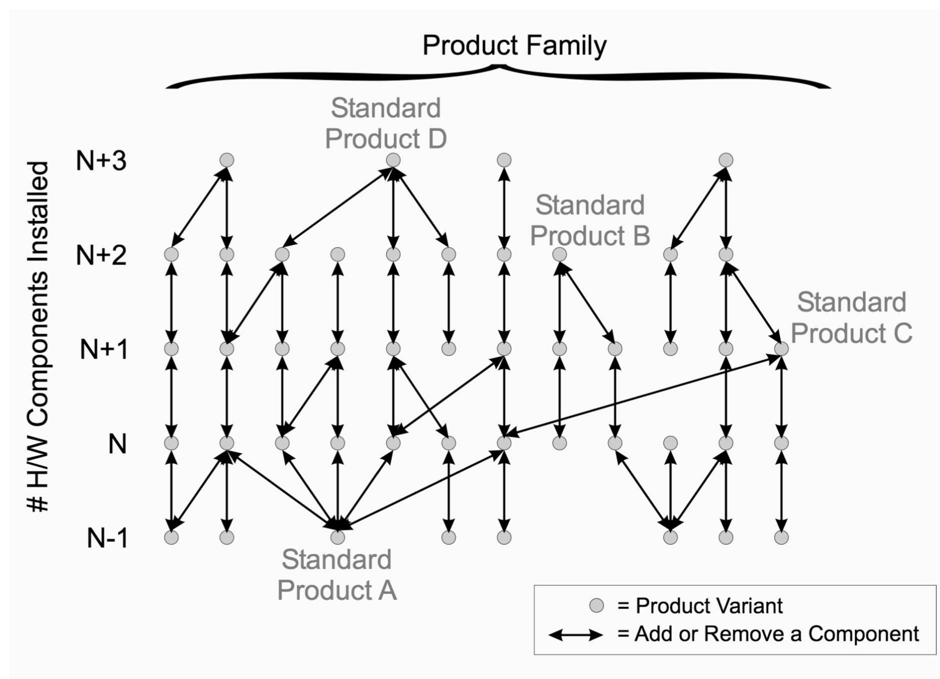


Figure 3.1: Example Lattice from [246]

[5] They call these components *adapters* and the library the *adapter repository*. The word *adapter* is motivated by the software existing in smart sensors converting raw sensor data to logical values required by the residual system, and software present in smart actuators converting logical control values to raw control signals that drive physical processes to affect the environment.

The more to the bottom of the lattice a vertex is placed, the less hardware components are intact, and quite probably the higher is the level of degradation of the system. The level of degradation is quantified by a so called *utility* value. The utility is a numeric value that represents the desirability of particular features. In order to minimize the level of degradation, the design objective is to maximize the *utility* of the system in the failure sequences, represented by the lattice vertexes. However, the work in [246] offers no guidance to the designer in generating these numeric utility values.

Compared to the work presented in this thesis, the lattice concept is similar to our notion of scenario graph, introduced in section 4.6.2. In the scenario graph, each transition represents the isolation of an execution unit or the isolation of a software component, due to assumed failures of these. Our notion of priority points of software components is similar to the utility values of features. However, we derive the priority points automatically from the ASIL criticality level and the fail-operational requirement of the functional features that are realized by the considered software component. Both the sum of priority points and the total system utility have to be maximized in order to minimize the level of degradation. We more focus on mixed criticality issues, on the expression of deployment constraints for valid redundancy schemes, on different types of redundancy by hot and cold standby slaves, as well as on required explicit deactivations of components to take care that fail-operational requirements of other components can be met by applying failovers. We also synthesize the configuration of communication channels between the software components, based on publish/subscribe port definitions. This enables to select those channels that lead to minimal network traffic. Moreover, we focus on a formal model and the expression of formal constraints and formal optimization objectives about all this. We apply an SMT solver with optimization capabilities to calculate solutions as basis for our analysis.

C.P. Shelton, P. Koopman (RoSES Project, CMU Pittsburgh, 2002 - 2004): In [312] [313] [314], the authors introduce an approach to analyze the graceful degradation of component based systems.[6] Some concepts are shared with the preliminary work from [246].[7] The intention is to build *implicit* graceful degradation into systems, without specifying failure scenarios a priori. They distinguish two criticality levels of components (critical and non-critical), two states of components (working and failed), and two significancies of functional features (primary and auxiliary). In contrast, we consider the ASIL classification for components (five levels), other component states due to redundancy (five states, Fig. 4.16), and we distinguish the functional features by their ASIL and required level of fail-operationality. We also calculate valid redundant deployments as part of our approach. When considering the relationship between software components and the functional features that are realized by the components, they distinguish the three types of *strong*, *weak* and *optional* dependency. In this thesis, we consider only strong realization relationships between software components and functional features, meaning that a functional feature cannot be provided at all, if any one of its realizing software components is lost.

Regarding communication dependencies between software components, they distinguish *required* and *optional* inputs. They treat optional inputs as *advice* to improve functionality when those inputs are available. We also distinguish mandatory and optional inputs, but we do not distinguish different levels of Quality of Service (QoS) of components depending on the availability of optional inputs.

They provide a *quantitative metric* of the systems ability to gracefully degrade, based on the notion of *utility* of system elements. The utility of the whole system and its sub-systems is calculated as *non-linear utility function* based on a boolean 0/1 utility of atomic components. If an atomic component has a failure, its utility is 0, else its utility is 1. The utility of the composed system elements is calculated based on the utility of the composition sub-elements, according to the utility function. The overall system utility is examined by considering the top-level feature subsets that provide outputs to the system actuators. The overall system utility is positive if and only if all of its critical feature subsets have positive utility. The

[6] `http://users.ece.cmu.edu/~koopman/projects.html#graceful`

[7] Both works were partially done as part of the project RoSES (Robust Self-configuring Embedded Systems), `https://users.ece.cmu.edu/~koopman/roses`

result is a *quantitative metric* for the graceful degradation of the overall system. The *utility function* is comparable to the sum of priority points that we introduce, but we apply a linear addition of the single priority points and consider redundantly active instances of software components in the sum.

To reduce the complexity of the consideration of the utility of different atomic components, they consider the system and software architecture and group components into subsystems based on the component interfaces. Then the subsystems are analyzed separately and the overall system utility is composed by the subsystem utilities. We group components by their ASIL and fail-operational requirements, enabling the separation of mixed critical components.

During the analysis, they use a discrete event simulator to inject faults to evaluate the reaction of the system to these faults and how the system gracefully degrades. We use a formal model with an integrated set of degradation scenarios to be analyzed, and harness an SMT solver with optimization capabilities to obtain solutions as basis for our analysis. They consider a fixed hardware configuration. We consider a hardware architecture whose provided resources decrease due to isolations of execution units. Like us, they do not consider fault detection mechanisms.

In contrast to them, we focus more explicitly on deployment constraints that ensure fail-operational behavior. We automatically obtain a valid redundant deployment of software to hardware, which fulfills the given fail-operational requirements, because the initial deployment and the degraded deployments are part of the result of a single combined problem. When analyzing the degradation scenarios, we apply failover mechanisms and explicit deactivations of components to keep alive fail-operational features. Furthermore, we support to model and analyze degradations on feature level, combined with diversity based degradations of software components.

P. Emberson, I. Bate (University of York, 2008-2009): In [114] [113], an approach for task allocation supporting graceful degradation in distributed embedded real-time systems is introduced. They reuse the utility function of [314] and integrate it into an optimization search function based on heuristic *simulated annealing* approach. The aim is to analyze how many *hot replicas* are required to ensure a certain utility function value in a fault scenario. In contrast, we assume a fixed amount of desired redundant instances in the deployment, depending on the required levels of fail-operationality of features. We distinguish hot and cold standby slaves. Instead of using a heuristic simulated annealing optimization, we use an SMT solver with optimization capabilities. They focus more on hard real-time schedulability analysis than we do, due to differences in the assumed system platform. We assume logical execution times and a RTE which handles master/slave failover switching, they consider bare-metal software and a scheduling model that requires to consider precedence relations. They do not consider optional communication channels and degradation chains on feature level with possible diversity of software components, but we do. Finally, based on the utility function, they provide a quantitative quality metric for fault-tolerance in different fault scenarios. We derive the level of degradation by the $available(f, \sigma)$ properties of functional features $f \in F$ in the degradation scenarios $\sigma \in \Sigma$.

M. Trapp et al. (Fraunhofer IESE, since 2003): In [334], a framework for fault-tolerance in safety-critical automotive systems is introduced, applying dynamic adaption as error handling technique. [8] Instead of using explicit predefined built-in redundancy, what produces additional cost, they make use of already present *implicit redundancy* in vehicle designs. They mention that for instance in typical vehicle designs, the *yaw rate* of a vehicle can be calculated in ten different ways without requiring any additional sensor. However, the differently calculated yaw rates may have different qualities! This means, if one sensor fails that delivers the yaw rate currently used by a specific function, another source of yaw rate can be used, but the function that uses the yaw rate then receives a less qualitative input value. Due to this,

[8]The work has been partially performed as part of the MARS project (Methodologies and Architectures for Runtime adaptive embedded Systems).

they design that each communicated signal is transmitted inclusive a quality information about the data that is encoded in the signal (e. g., the quality of the yaw rate). To handle this, the authors model functions with different degradation levels. As functions contain software components, also the components have different degradation levels (also called different configurations). With the information about quality of input data, the transitions between degradation levels of functions and components can be specified. If an error appears leading the quality of input data to decrease below a threshold, the functions degrade that require the input data in a quality above the threshold in their current configuration, such that the new degraded configuration is able to work with the less qualitative input. Non critical functions can be completely switched off in case of errors.

Compared to their contribution, in this thesis we introduce a more detailed formal model than the relatively abstract formal model introduced in [334]. We distinguish mixed-criticality levels (by ASIL) and mixed reliability requirements in form of different required levels of fail-operationality. Based on our formal model, we are able to statically analyze degradation scenarios simultaneously and synthesize deployments that lead to optimal analysis results. Our model also supports the definition of degradation levels of functional features and software components. We also consider deactivations of features with no or low enough fail-operational requirements. However, we focus on explicit redundancy of software components. We calculate optimal levels of redundancy during our deployment synthesis, achieving that cost by redundancy does not become higher than necessary. We do not focus on implicit redundancy by multiple sources for a piece of data like done in [334], but by modeling an input model with multiple publications with identical *data-ID* (see section 4.4.2). By adding a quality information for each *data item*, we could formally model something very similar and then statically analyze the considered failure scenarios. Although our formal model currently does not contain a quality information about communicated pieces of data, this could be added easily. However, for this the communication matrices, introduced in section 4.2.2 Def. 6, would need to be extended to be matrices over the union set of ports, not over the set of software components, as the quality property would be required on port level to distinguish different qualities of different subscribed data of a component.

M. Glass, J. Teich (University of Erlangen-Nuremberg, 2009): In [136], the authors also tackle the design of gracefully degrading mixed critical systems. They focus on a degradation-aware reliability analysis, also considering redundant deployments. The objective is to maximize the systems reliability in different degradation modes, in which different levels of functionality are provided based on the residual sets of intact resources. Instead of optimizing the reliability in the degradation modes separately in a multi-objective manner, they offer a single objective approach in which the designer can assign weights to the different degradation modes to control how much the modes influence the objective. Reliability is measured by the metric of *Mean Time To Failure (MTTF)* as an integral over the time. They consider applications that consist of multiple tasks, what is in principle similar to our functional features that are realized by software components. During design time, the degradation modes are predefined and stored into *Binary Decision Diagrams* (BDDs). Critical tasks may be deployed to multiple execution units. In the model, for each degradation mode they mark which resources are defect in the mode, which is similar to our $isolated(e)$ property of execution units $e \in E$. During runtime, a dedicated reliable *observer* component is able to detect failures and uses the BDD data to decide which task instances have to be activated or deactivated in which execution unit in case of a resource failure. The idea behind this is similar to our objective, namely being able to deactivate low critical tasks to provide opportunities to keep alive high critical fail-operational tasks. Like in our work, they do not focus on how to detect failures, as it is not explained how the observer component does this.

M.P. Herlihy, J.M. Wing (Cambridge Research Laboratory and CMU Pittsburgh, 1987-1991): In [162] and [163], the authors introduce an approach to specify the desired preferred behavior of programs, as well as relaxed degraded behaviors that may become necessitated in case of environmental changes at

runtime, like faults, timing anomalies, synchronization conflicts or security violations. The introduced method is called *relaxation lattice*. Each node in the lattice describes a set of constraints. In the root node, all constraints are fulfilled and the preferred behavior is provided. In case of environmental changes, it may be the case that some constraints cannot be fulfilled anymore, leading to a transition to a another node in the lattice and a degraded behavior of the program. Each constraint has an associated cost. The higher the sum of the cost of fulfilled constraints, the closer the actual behavior is to the preferred behavior. The aim is to describe the resulting relaxed behaviors by using axiomatic specifications. The system model encapsulates sequential threads of control (processes) that concurrently access data objects. Data objects are defined by a type and primitive manipulation operations. The system computation is modeled as a history sequence of interleaved operations on objects. An example is given for instance by describing degraded behaviors of a replicated priority queue, based on two constraints and a resulting constraint-lattice of $2^2 = 4$ nodes. In difference to their approach, we focus on a structural analysis of degraded architectures, not on describing degraded program behaviors. We focus more on mixed-criticality systems and determine required failovers to redundant backup software components, as well as required explicit deactivations of software components, to ensure that fail-operational features can be kept available. On technical level, the relaxation of constraints is similar to the soft-constraints that we apply in this thesis (see section 4.6.7). However, they use relaxed constraints to describe degraded relaxed behavior, while we use soft-constraints to identify parts of the constraint set that cannot be fulfilled by the current architecture, giving a hint about parts of the architecture that have to be changed in order to fulfill all constraints.

SafeAdapt Project (Fraunhofer ESK et al., 2013-2016): In [263], the authors introduce a meta-modeling approach to describe architectural patterns for fail-operational, gracefully degrading systems.[9] Different redundancy patterns and graceful degradation patterns are listed from literature. A so called fail-operational graceful degradation (FOGD) pattern is introduced and incorporated into the pattern meta-model library. The FOGD pattern is inspired by an existing state decrement pattern [300]. Decrementing a state supposes that system features have different admissible states beside their full-fledged state. This is comparable to our approach to model degradations on feature level, just that we do not call a degraded feature a state. Our description of a feature degradation in section 4.7 corresponds to the state transitions. In addition, they support to describe entry and exit actions for states, like notification of the driver in case of a degradation. The applicability is demonstrated based on two examples, an automotive brake-by-wire example incorporating redundancy with hot-standby slaves, and a speed control example showing one graceful degradation scenario based on the FOGD pattern. However, they do not apply a metric to measure the degree of degradation that the systems experiences in different failure scenarios. Nevertheless, the work might be usable to create software architectures as input model for our approach, enabling our approach to analyze degradation scenarios of architectures incorporating espoused redundancy and degradation patterns, like redundancy by master and hot/cold-standby slaves. Deployment (allocation) of software to hardware is not considered and mentioned as possible future extension.

In [290], a so called *generic adaption mechanism* (GAM) is introduced as functional safety concept to enable fail-operational behavior in automotive systems. Hot-standby and cold-standby instances of application components are Degradations of applications are considered in case of failures of system elements. For instance, a steer-by-wire application exist in a full-functional primary version, as well as in a more basic degraded version. Initially, the primary is active and the degraded version is present as hot-standby slave on another execution unit. After a failure, the primary version may become lost and the degraded version may be used. Our formal model of feature degradations presented in section 4.7 can be used to describe these degradations of applications. Afterwards, our approach can be used to formally analyze the different degradation scenarios after assumed failures of system elements. However, our approach enriches this by an automated synthesis of optimal redundant deployments with cold- and hot-

[9] The work has been performed as part of the SafeAdapt Project, `http://www.safeadapt.eu`

standby slaves, incorporating the analysis of required failovers in failure scenarios to hold fail-operational requirements, as well as an analysis about the required level of graceful degradation of non fail-operational features due to insufficient resources in failure scenarios due to isolations of failing system resources. Finally, the work from [290] and the work presented in this thesis might be used complementing each other to design fault-tolerant, gracefully degrading mixed-criticality systems.

SAFER Approach (CMU Pittsburgh and General Motors, 2012-2013): In [197], a system architecture for dependable autonomous vehicles is introduced, also supporting graceful degradation in failure scenarios. [10] The considered SAFER architecture [196] supports fault-tolerance by using redundancy mechanisms based on cold standby slaves, hot standby slaves and task re-executions. They also consider graceful degradation of the system, for instance in case of failures of processor boards. They consider that the graceful degradation of vehicles should be appropriately adjusted depending on different situations. For instance, if a vision algorithm for pedestrian detection fails for instance due to a failure of a microcontroller, the reaction should be different when driving on a highway, compared with driving in an urban area, as pedestrian are more likely to be present in the latter case. We do not consider such environment dependent (or context aware) degradation strategies in this thesis, but for instance based on the system mode concept sketched in future work section 6.3, or based on another kind of extension, our formal model and analysis approach could be geared up to support such scenarios. As type of degradation they consider the reduction of the utilization (the ratio of WCET and period) of tasks on a processor by prolonging the execution period of tasks. By this, tasks are executed less often, resulting in a degradation of the quality of service - for instance of image recognition algorithms or closed loop control algorithms. They call this adaptive resource management. We do not consider such degradations of execution periods (alias cycles) in this thesis, as we assume a single rate scheduling for simplicity. However, the presented SAFER architecture [196] can be seen as a platform that is to a huge amount compatible to the analysis approach introduced in this thesis. After introducing the sketched extensions to our formal model and adjusting the deployment constraints to SAFER specific requirements, our approach could be used to synthesize valid redundant deployments for SAFER based systems, as well as analyze degradation and failover scenarios. We assume a system having appropriate runtime failure detection and failure isolation mechanisms, which is fulfilled for instance by the RACE approach (see section 2.5), but is claimed also by the SAFER approach.

Other work on Design and/or Analysis of Graceful Degradation: In [59], the *Architecture Analysis and Design Language* (AADL) [295] is used to model a systems nominal and faulty behavior. AADL supports to model different operating *modes* for model entities (like devices), as well as related *mode transitions*. They use AADL to model *degraded modes* of operations and specify mode transitions by mode transition guards and mode transition effects. One focus of their work lies on a detailed specification of the operational behavior of components, particularly covering hybrid systems with continuous and discrete values. The authors introduce a dependability analysis approach for AADL models comprising degraded modes. They also analyze the modeled system w.r.t. performance requirements by applying probabilistic model checking techniques. In contrast to their work, in this thesis we do not model the operational behavior of components, and we do not apply probabilistic or stochastic methods to describe the characteristic of fault appearance. Instead, we focus on synthesizing optimal redundant deployments of mixed-criticality software to hardware for different scenarios that may appear during system run-time. We ensure the validity of the deployments by formalized constraints. The scenarios cover isolations of failed execution units and software components, as well as failover mechanisms ensuring that fail-operational requirements are hold. Our approach allows to analyze which functional features the system will be able

[10] The work has been performed as part of the SAFER project (System-level Architecture for Failure Evasion in Real-time Applications) at CMU Pittsburgh

to provide in which scenarios, returning also the level of degradation that the available system features will undergo. We apply an SMT solver to generate solutions for the formalized problems.

In [56], a degradable safety controller for a steer-by-wire application is presented, using an on-chip redundancy by using multiple cores. For instance, based on a quad-core CPU, a 1-out-of-4 (1oo4) redundancy is introduced. If one core fails, this is degraded to a 1oo3 redundancy, etc. This increases the availability of the steer-by-wire application that is executed on the chip. However, it remains unclear if also the quality-of-service (QoS) of the steer-by-wire application is degraded in case of a failure of one of the chips. This seems not to be the case. They do not tackle the problem of how to deploy software mixed redundantly to these cores and how to deactivate software parts to keep available those software parts having fail-operational requirements.

In [149] and [150], graceful degradation is tackled for the smart energy domain, to express degradations of powered electrical devices in smart buildings, in case of power outages leading to a battery driven operation mode of a building. The intention is to decrease the set of powered electrical devices to make efficient use of the battery capacity. The approach is also constraint based and uses the Z3 SMT solver.

More work towards graceful-degradation had been done for instance in [44] (addressing fault tolerant computing with graceful-degradation, but not considering separation of mixed-criticality components and not considering deployments), as well as in [140] and [315].

3.1.2 Design and Analysis of Reliability and Robustness

In this section, we discuss related work in the area of designing and analyzing reliable and robust systems.

In [153], an iterative design space exploration (DSE) approach is introduced with focus on system robustness and performance. The approach is explicitly designed to support the design space exploration of a big system in form of independently optimizing parts of the system developed by different design teams, and iteratively combine the results and obtain a design of the whole system. The motivation behind this is that often no single design entity has all information to design and optimize the whole system at once. They focus on the optimization of the robustness of embedded systems with respect to variations of system properties. It is mentioned that this is a meta problem that does not directly arise from the expected and specified functional system behavior. Considered robustness metrics are the *static design robustness* (SDR) metric, the *dynamic design robustness* (DDR) metric, as well as the *robustness gain*. The SDR and DDR metrics are distinguished for properties with and without performance dependencies. In case of dependent properties, modifications of one property influence the flexibility of other dependent properties. To tackle the computationally expensive multi-dimensional analysis for interdependencies, they introduce a scalable stochastic analysis method, approximating system robustness in a multi-criterion optimization problem. With the robustness gain metric, the benefit of designing reconfigurable system components is explicitly measurable. Redundancy or replication mechanisms re not taken into account. In contrast to [153], in this thesis we treat robustness in the classical form of robustness against permanent faults, assuming an isolation mechanism for faulty system elements. We calculate redundant deployments and failover strategies to provide fault-tolerance in different scenarios, while ensuring that functional features with fail-operational requirements are kept available. We measure the level of degradation of the system in the scenarios by identifying the functional features that cannot be kept available, for instance due to insufficient resources. They utilize a multi-objective evolutionary algorithm. We apply an off-the-shelf SMT solver to perform our analysis, conducting a multi-objective optimization to calculate the synthesized deployments. We do not apply approximation or heuristics to trade-off efficiency and optimality.

Reliability-driven deployment optimization for embedded systems has also been presented in [237] [238]. The aim of [238] is to bridge the gap between deployment-targeting approaches and reliability models, resulting in a reliability-driven approach to the system deployment problem. The reliability is taken into account by including hardware-level reliability approaches, and formalizing the propagation of

hardware-level reliabilities to the reliabilities of software-level services. A DSE is done using evolutionary algorithms. In [237], the problem of uncertain design time parameters, leading to sub-optimal design decisions, is tackled. They introduce an optimization approach that deals with parameter uncertainties at design time and finds solutions that restrict the impact of parameter uncertainties to provide better decision support.

In [146], architecture trade-off analysis for conflicting quality requirements is tackled by using an evolutionary algorithm and multi-objective optimization strategies, based on architecture refactorings. The approach aims to reduce development cost and improve the quality of the system design. As case-study, they use a satellite system with robustness requirements and with redundancy and fault detection mechanisms, but analyze only a simplified architecture without any redundancy. The benefit of using an evolutionary algorithm is the scalability to be able to handle large scale problems. In contrast, in this thesis we focus on synthesizing redundant deployments and analyze degradation scenarios after assumed failures on a structural level, without setting scalability of problem solving into foreground.

The work in [239] introduces an environment for a flexible and tailorable specification, manipulation, visualization, and (re)estimation of deployment architectures for large-scale distributed systems. They tackle the problem of deploying (alias allocating or mapping) interacting software components to hosts. The objective is to find a deployment that maximizes the system's *availability*. They define availability as the ratio of the number of successfully completed inter-component interactions to the total number of attempted interactions over a period of time. They investigate six algorithms tor increase availability by calculating new deployments, named 1) exact, 2) unbiased stochastic, 3) biased stochastic, 4) greedy, 5) clustering, 6) decentralized algorithm. Redundancy or replication is not considered by their approach. Therefore, they cannot analyze failover scenarios, what is one of the key research questions of this thesis. Another contrast is that we consider availability of functional features only at a certain point in time, what is for sure an unusual definition, but could be extended to the usual ratio based definition of availability by introducing a notion of time to our model, what is out of scope of this thesis. We also consider interactions between components by communication channels.

Reliability and Robustness of Systems with *Redundancy*: In [224], the trade-off between mission reliability, mission cost and mission time is tackled. The design objectives are to maximize mission reliability, while minimizing mission time and cost. They solve this multi-objective DSE problem by applying a genetic algorithm (GA). They also consider backup procedures and their influence on mission time and mission reliability in case of failures, using *cold standby* computing systems with periodic backups.

In [84], a *redundancy allocation problem* for cold-standby redundancy is tackled for non-repairable systems. Their objective is to select from available components and to determine an optimal design configuration to maximize system reliability. They consider component time-to-failure distributed according to an Erlang distribution. Optimal solutions are determined based on an equivalent problem formulation and integer programming. In [85], the work from [84] is extended by determining optimal solutions to the redundancy allocation problem, when either active or cold-standby redundancy can be used. The objective is the selection of components and redundancy levels to maximize system reliability. No predetermined redundancy strategy is given, instead the choice of redundancy strategy becomes an additional decision variable. Optimal solutions to the problem are found by an equivalent problem formulation and integer programming. Further related work had been performed in [89], considering a combination of cold standby redundancy and active redundancy.

In [109], a concept to enable the efficient use of remote redundancy for safety-critical systems is presented to create fault-tolerant or fail-safe applications. They introduce a *signature-protected communication* to integrate redundant peripheral into a system. They propose to connect redundant peripheral to the most proximate control computer of the network, reducing wiring harness without compromising fault-tolerance. As their work enables the decoupling of function and location, remote redundancy can

be shared between different subsystems. They claim that communication delays can be neglected, when using a sufficiently fast communication system. As result, they show that remote redundancy allows for a significant cost reduction for redundant hardware without compromising fault tolerance characteristics. Compared to their approach, in this thesis we consider a similar system design, also with decoupling of function and location, enabling a flexible deployment with a flexible degree of redundancy. We assume a system architecture similar to the one proposed in the RACE project (see section 2.5). In contrast to the work shown in [109], we incorporate the synthesis of deployments of mixed-criticality components with an optimal degree of redundancy (with hot or cold standby slaves), to fulfill mixed reliability requirements of functional features in form of different fail-operational requirements. We express mixed-criticality separation requirements by a notion of clusters and related deployment constraints. Finally, we analyze the synthesized deployments for different failure scenarios with respect to the degree of functional degradation that the system undergoes. The degradations become necessary due to insufficient resources to keep the full set of features available, caused by assumed failures and subsequent isolations of execution units. We perform our analysis based on a formal system model, formal constraints, and formal optimization objectives. In [109], the creation of a formal model about their work is mentioned as a future work.

In [183], the authors consider two fault-tolerance techniques: 1) re-execution of processes (time-redundancy), and 2) active replication (space-redundancy). They show how re-execution and active replication can be combined in an optimized implementation that leads to a schedulable fault-tolerant application without increasing the amount of used resources.

A design framework for reliable high quality execution of closed-loop control systems is presented in [298]. In case of failures of computation nodes (alias execution units), the aim is to adapt the system to achieve fault-tolerance and keep alive the control tasks. They model sequences of node failures as *Hasse diagram*, like we do it in this thesis to model our scenario graph (section 4.6.2 and 4.6.10). To achieve the fault-tolerance, they consider trade-offs between replication of tasks to multiple processors, and dynamic task migrations between processors in case of failures. They determine configurations comprising mappings (alias deployments) of tasks to processors, as well as schedules of the processors. However, as the configuration space due to computation node failures is exponential, they suppose that it is sufficient to synthesize only a small amount of base configurations to achieve fault-tolerance with an inherent minimum level of control quality. To improve the control quality, they propose an algorithm for a priority-based search of configurations. In this thesis, we do not consider dynamic task-migrations, but we synthesize appropriate levels of redundancy (by deploying cold- and hot standby slaves) to achieve fail-operational requirements in mixed-critical, mixed-reliable systems. We formally analyze degradation and failover scenarios for failures of execution units and software components, while synthesizing deployments for each scenario ensuring that fail-operational requirements are met and that the level of degradation is minimized.

The work presented in [349] tackles the problem of achieving reliability for service composition problems, being the problem of flexible integration of service functionalities with achievement of sufficient reliability. The considered design task is to adapt workflows to the non-functional requirements of the user, what becomes a complex problem with increasing number of services in a workflow, such that previous approaches fail to achieve a sufficient reliability. To enable reliability in case of service failures, they integrate *backup* services. Their approach includes the computation of a quality of service (QoS) optimized selection of service clusters that includes a sufficient number of distributed *backup* services for each service, to prevent loss of service instances - for instance due to a network failure. In addition, they explicitly consider the costs of planning and replanning the workflows. They setup a multi-objective problem that considers the possible *repair costs* directly in the initial service composition. The resulting QoS of the workflow and the location of the services is visualized to the designer to enable him to select compositions with risk preferences. In contrast to their approach, in this thesis we do not consider the problem of designing workflows of services. We consider the deployment of mixed-criticality software components to embedded execution units of a system, incorporating an optimal level of redundancy by usage of hot or cold standby components. We focus on the analysis of failure scenarios, enabling the

synthesis of deployments taking care that fail-operational requirements of functional features can be met by keeping these those components alive that realize these features. Simultaneously, we minimize the degree of functional degradation appearing due to required deactivations of functional features without fail-operational requirements due to insufficient execution resources in failure scenarios.

3.1.3 Design and Analysis of Availability

An approach to specify and analyze the *availability* of software intensive embedded systems is introduced in [186]. The author introduces an artifact model to express availability requirements. The *behavior* of the system is modeled based on the FOCUS theory [69], enriched with availability metrics. Also a modeling guideline is introduced about how to create the availability specific model artifacts. The analysis is based on using the probabilistic model checker PRISM. In contrast, in this thesis we treat availability of functional features only at certain points in time, not over an interval in time. We do not model the functional behavior of software components, but instead provide a structural analysis about which functional features can be kept available in which failure scenarios, by analyzing insufficiency of system resources to execute realizing software components. We combine this with the synthesis of optimal deployments of mixed-criticality software components to hardware execution units and the synthesis of communication channels between the components, based on publish/subscribe definitions of communication ports. This combination of synthesis and analysis enables to find an deployment architecture that is optimal in terms of allowing to fulfill all fail-operational requirements of functional features, while keeping the degree of functional degradation in failure scenarios as low as possible, and not integrating more redundancy than required to fulfill the fail-operational requirements. Instead of applying an probabilistic model checker, we apply a SMT solver to calculate solutions for the modeled problems.

3.1.4 Fault-Tolerant Scheduling for Mixed Criticality Systems

In [299], fault-tolerance for mixed criticality multiprocessor systems is enabled by dynamic task migrations between processors in case of permanent processor faults. Mixed safety criticality is considered by distinguishing tasks that have requirements to survive transient processor faults, permanent processor faults, or tasks that have to fault-tolerance requirements. Also mixed time criticality is considered, distinguishing soft and hard real-time deadlines and by using *Earliest Deadline First* (EDF) scheduling for hard real-time tasks, and *Constant Bandwidth Server* (CBS) scheduling for soft real-time tasks. CBS enforces temporal isolation between hard and soft real-time tasks. The migration decision is done dynamically at runtime in case of a detected permanent processor fault, using a *greedy-based online heuristic*. In case of decreasing resources due to permanent processor faults, performance degradation of soft real-time tasks may appear. The objective is to maximize the quality of service (QoS) (the performance) of soft real-time tasks by maximizing the probability that soft deadlines are met. Transient processor faults are handled using checkpointing and rollback recovery. In this thesis, we focus on an static design time analysis and synthesis approach for fault-tolerance in mixed-criticality systems. Instead of calculating and performing task migrations at runtime in failure scenarios, we pre-calculate adequate levels of redundancy to match fail-operational requirements in combination with a failover mechanism. Even although redundancy always introduces a certain level of overhead and cost, our approach supports to construct a system with a predefined range of dynamism without requiring task migrations, that can be analyzed at design time, supporting to certify a safety critical fault-tolerant system at its design time. We do not consider soft real-time tasks or the analysis of their deadline exceedances. Instead of using a heuristic optimization approach at runtime, being a trade-off between exactness and calculation time, we use an exact optimization approach at design time, based on an off-the-shelf SMT solver.

The work in [332] tackles fault-tolerant fixed priority scheduling of mixed criticality task-sets, having hard and soft deadlines. The fault-tolerance of critical tasks is ensured by re-executions of faulty critical

tasks by replicated alternate tasks on different processing nodes, within the deadline of the original faulty task. They provide hard real-time fault-tolerance guarantees for critical tasks offline (at design time), and ensure flexibility for non-critical tasks online (at runtime) by maximizing the resource utilization for non-critical tasks. They calculate the allocation (alias deployment) of tasks to processing nodes (alias execution units), and derive so called *feasibility windows* as well as scheduling attributes, like priorities of tasks. They apply *integer linear programming* (ILP) to derive the scheduling attributes. One objective is to maximize the resource usage by splitting tasks into so called artifact tasks, while minimizing the amount of such task splits. In this thesis, we focus on the calculation of an optimal level of master-slave redundancy in the deployment of mixed-criticality software components to execution units, to ensure fail-operationality of functional features in certain failure scenarios. We analyze the level of required degradations of the set of available functional features in the failure scenarios, and also support to model and analyze degradations on feature level, to substitute a full-fledged functional feature by a degraded version of that feature, if the full-fledged feature cannot be kept available anymore, for instance due to insufficient resources after a failure. We do not consider priority based scheduling and do not distinguish hard and soft deadlines. As problem solving technology, we use an SMT solver with optimization facility.

3.1.5 Design of Structural and Behavioral Reconfiguration

In [75], a formal design approach supporting structural reconfiguration and behavioral adaptation in fault-tolerant systems is introduced. They present a formal model of system configuration and reconfiguration contracts, based on a *Labeled Transition System* (LTS). For instance, this allows at predefined states to reconfigure a component by another one that implements a different behavioral interface. However, they do not consider mixed-criticality systems and do not tackle the deployment/allocation problem of software to hardware. Redundancy or replication mechanisms to ensure fail-operational architectures are also not in their scope.

In [33], the authors introduce an approach for architectural online reconfiguration in AUTOSAR based automotive systems, to tackle robustness and flexibility in case of failures of system elements (like sensors). Reconfiguration models are added to typically fixed AUTOSAR models, to describe for instance alternatives for connectors between software components. This is used in an example to model alternative connectors in case of failing sensors, to model an interpolation of the value of the failed sensor based on the values of the intact sensors. To manage the connector reconfigurations at runtime in a fixed AUTOSAR architecture, a software component named "Reconfiguration" is added to the architecture and the modeled alternatives are transformed back into a merged architecture. The Reconfiguration component has all alternative connectors as input and internally routes the signals appropriately to the outputs. However, addition or removal of software components is not possible. Furthermore, they do not tackle mixed-criticality systems, redundant deployments of software components, or failover mechanisms to ensure fail-operational requirements. They manually model the deployment of software to hardware and do not apply an automatic synthesis of valid deployments with appropriate redundancy to meet fail-operational requirements in different failure scenarios. All of this is tackled in this thesis.

Dynamic Architecture Description Languages (dynamic ADLs): On the level of software architecture, there had been a lot of research about dynamic software architectures and corresponding dynamic software architecture description languages (dynamic ADLs) in the late 90s. The idea was to describe possible architectural evolutions in the design phase, like addition or removal of components, ports or communication channels, that are allowed to appear during runtime. Surveys and classifications over the different approaches are given in [256], [236], [61] and [257].

In this thesis, we analyze dynamic aspects of software architectures with respect to the question about which instances of redundantly deployed components can be kept active in which failure scenarios, which

instances have to be deactivated due to insufficient resources, and which redundant backups have to be activated to perform a failover to ensure fail-operational requirements. However, this does not mean to describe evolutions of the software architecture. The set of components in the software architecture itself is fixed, but it is dynamic which instances of the components are active on which execution units. We also synthesize the established communication channels during our analysis, connecting the active component instances. If an active instance fails or is lost due to an isolation of an execution unit, another redundant instance becomes activated on another execution unit, if required. We assume an underlying RTE that ensures a correct routing of the communication channels, and therefore a correct transmission of data between the active software component instances. The set of communication channels itself is fixed. Only changes of the data routing may appear in failure scenarios, if a failover is performed. No new channels are added between components during the failure scenarios. Hence, we do not describe explicitly the addition and removal of communication channels, like it is done in many dynamic ADLs (e. g., see Table 3 in [61], channels are called *connectors* there). Differently from being done in dynamic ADLs, we also do not explicitly describe possible evolutions of the deployment of software to hardware, but we synthesize possible valid deployment evolutions during our analysis. A dynamic ADL may be usable to encode allowed runtime deployment reconfigurations for the system under analysis, containing reconfigurations that are valid according to the result of our analysis, but this is out of scope of this thesis.

Dynamic Software Product Lines: Another approach to describe architectural runtime reconfigurations are *dynamic software product lines* (dynamic SPLs) [152]. Software product lines describe a set of product variants that can be created from a common product line, based on reuse of components and specifying which components differ between product variants instantiated from a product line. The functional features of a product line, as well as the possible reconfigurations of the feature set when switching between product variants, can be designed for instance in a dynamic feature model [101] . Orthogonal Variability Models (OVM) [316] and the Variability Modeling Language (VML) [226] are other notations to model product line variability. In classical static SPLs, the variant decision is done at design time and the corresponding software architecture is fixed and does not change at runtime. In dynamic SPLs, switches between product variants are possible at runtime, performed by reconfigurations of the software architecture, like replacements, additions or removals of software components and connectors. Architectural adaptation mechanisms for dynamic SPLs are discussed in [77]. However, as the same for dynamic ADLs, there is no built-in focus on synthesizing valid deployments of software to hardware, on ensuring fault-tolerance and fail-operational requirements by using appropriate redundancy and failover scenarios, or on analyzing the degree of degradation that may be necessary in failure scenarios. Finally, dynamic SPL approaches may be usable to encode degradations of the functional feature set that are identified as necessary by using the analysis approach introduced in this thesis.

3.1.6 Self-x Approaches

After the investigation of dynamic ADLs in the late 1990s and early 2000s (see section 3.1.5), a new research field opened towards dynamic systems with *self-x* (alias *self-**) properties. A lot of approaches exist to apply self-x techniques to create dependable embedded systems, partially incorporating graceful degradation of the system in case of internal failures of system elements. In this section, we provide a brief overview over fundamental terminology, concepts and some applications.

Terminology: We now give a brief overview about the major self-x properties that are often used to create fault-tolerant or gracefully degrading systems. We focus here on *self-adaptation*, *self-organization*, *self-configuration* and *self-healing*. Many more self-x properties got defined, which are either synonyms or similar (like self-repairing alias self-healing), or related properties (like self-managing or self-optimizing), being either sub-properties or super-properties of the above mentioned properties. The works in [296],

[285], [195], and [244] provide good starting points to get an overview over all the already defined self-x properties and their relationships, which we do not completely list here.

Self-Adaptation is defined in many, but similar, fashions. To name just a few:

- A self-adaptive software system is one that can modify itself at run-time due to changes in the system, its requirements, or the environment in which it is deployed [12].
- A self-adaptive system evaluates its own behavior and changes behavior (or performance) when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible [297].
- Self-adaptivity is the capability of a system to adjust its behavior in response to the environment [231].
- Self-adaptive systems work in a *top-down* manner. They evaluate their own global behavior and change it when the evaluation indicates that they are not accomplishing what they were intended to do [310].

Respective the question about what becomes adapted (or reconfigured), the work in [7] distinguishes 1) parameter adaptation, 2) structural adaptation and 3) behavior adaptation. The work in [123] distinguishes the adaptation of 1) structure, 2) functionality, 3) parameter, 4) content and 5) resources. In this thesis, we focus mainly on resource adaptation. *Resource adaptation* means the dynamic allocation of resources, in our case the changes of the deployment state (active, passive) of software to hardware in the analyzed failure scenarios. The re-allocated resource is the computing capacity of the processor of the execution unit, as well as the network data transmission capacity due to possible changes in network traffic. *Structural adaptation* means for instance the dynamic replication of objects or the addition of new software modules during run-time. These structural adaptation mechanisms are not in focus of this thesis.

Self-Organization is defined in [97] as *bottom-up* spontaneously creation of a new organization of system entities in case of environmental changes without external control. Self-organization is motivated by swarm intelligence in nature (ants, termites, honey bees, etc.) [97]. In [244] it is defined that a system is *self-organizing* if and only if it is self-managing, structure-adaptive, and employs decentralized control. Self-organization exists with and without *emergence* of higher level properties based on decentralized lower level decisions [94] [97]. Strong and weak self-organization are distinguished in [97]. Strong self-organizing systems work without any explicit central control from internal or external. Weak self-organizing work with some internal central control or planning.

The *bottom-up* aspect of self-organizing systems is the major difference to self-adaptive systems, which are *top-down* approaches [81]. However, also combinations of top-down and bottom-up approaches exist [310] [326].

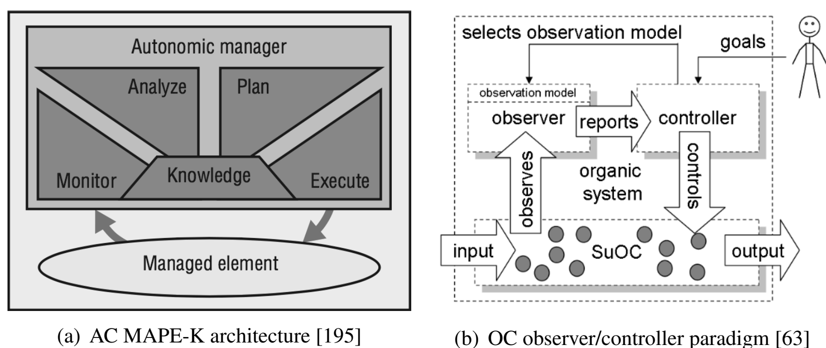
Self-Configuration establishes an automated configuration of components and systems following high-level policies. The rest of the system adjusts automatically and seamlessly [195]. One approach to enable self-configuration is for instance presented in [323]. The deployment synthesis that we introduce in this thesis is also a kind of automated configuration. However, the major use case that we consider is the application at design time. The analysis results can be stored and used to construct pre-configured dynamism, or to analyze the decision space of decision mechanisms implemented in the system itself. We do not focus on self-configuration at runtime by the system itself, but when using a more efficient heuristic calculation of solutions, our analysis could also be performed at runtime, contributing to establish self-configuration.

Self-Healing is defined as the ability of a system to monitor and heal itself from the inside, which requires the ability of this system to decide about and perform recovery actions to return itself to a behavior conforming to its initial specification, especially without external interference [285]. Other nearly synonymous terms for self-healing are self-repairing, self-regeneration and self-immunity [285]. To be self-healing, systems need reflective capabilities to perform introspection by monitoring the running state and identify anomalous behaviors. When any failure is detected, some repair actions have to be performed to recover form the failure [311]. Koopman defines in [206] that self-healing systems might not restore complete functionality after a fault, allowing degradation, and that the degree of degraded operation provided by a self-healing system is its resilience to damage that exceeds built-in redundancy. Sometimes this is seen as the difference to *fault-tolerance*, e. g., [285] defined that fault-tolerance aims at keeping the system running at 100% of its designed functionality, while self-healing can mean that after the healing the system operates at less than 100%, meaning that the system degrades. In contrast, the work in [23] states that self-healing and fault-tolerance are synonyms, and that further synonyms of fault-tolerance are self-repair and resilience. In this thesis, we follow the view of [23] and allow fault-tolerant systems to degrade, as long as no fail-operational requirements are violated.

Research roadmaps with respect to self-adaptive systems are given in [296], [81], [92] [231] and [243].

Autonomic and Organic Computing: Autonomic Computing (AC) and Organic Computing (OC) are approaches to create self-organizing and self-healing systems.

- *Autonomic Computing* (AC) was introduced in [195], together with the MAPE-K loop pattern (Monitor, Analyze, Plan and Execute - based on Knowledge) of autonomous systems (Fig. 3.2(a)). AC is fully autonomous and AC systems can manage themselves given high-level objectives from administrators. In [195] this is called self-management, composed by the abilities of self-configuration, self-healing (error detection, diagnostic, repair), self-optimization (of parameters) and self-protection (against attacks). AC systems are context-adaptive systems [201] with awareness about the system environment.
- *Organic Computing* (OC) was introduced in [306], allowing external user influence to perform a controlled self-organization [307]. A *observer/controller* paradigm (Fig. 3.2(b)) is described to establish self-organization of a system under observation and control (SuOC), being applicable in a centralized, distributed or hierarchical multi-level manner [63].



(a) AC MAPE-K architecture [195]

(b) OC observer/controller paradigm [63]

Figure 3.2: AC and OC self-organization loops

Twelve design patterns for dynamically adaptive systems are presented and discussed in [279], tackling monitoring, decision making and reconfiguration. Examples for reconfiguration patterns are component insertion, component removal, reconfiguration of continuously available servers, as well as decentralized reconfiguration.

Applications of self-x techniques to construct dependable systems: Both top-down self-adaptive approaches, as well as bottom-up self-organization approaches (AC or OC), can be used to create reliable fault-tolerant (respectively self-healing) systems. As mentioned above in the terminology part, often self-healing and fault-tolerance are distinguished by the view that self-healing allows degradation of systems after failures of system elements, and fault-tolerant systems do not allow degradation. However, in this thesis we do not follow this view and allow fault-tolerant systems to degrade, as long as no fail-operational requirements are violated.

In context of the project DySCAS [11], the work in [120] considered dependencies between tasks and the determination of executable tasks based on available input data. If input data is missing for a task, the task is deactivated. The procedure is repeated iteratively until no more tasks have to be deactivated. They present a distributed algorithm without requiring central knowledge. The considered system uses *Rate Monotonic* (RM) scheduling. Tackled use cases are system startup, attaching devices during runtime and the reaction to hardware errors. The deployment of tasks to execution nodes is assumed to be already determined. Redundancy in deployment is not considered. Instead, online task migrations are assumed. Task sets with mixed priorities are considered and the most important schedulable subset is identified. However, explicit degradation based on mixed safety criticalities is not tackled.

In [100] [99], the deployment problem of dependent components to a heterogeneous automotive network is modeled as *Constraint Satisfaction Problem* (CSP) and solved with a *Backtracking* or a local *Iterative Repair* algorithm. They aim for a centralized self-management with focus on self-configuration and self-healing. As use cases, updating, installing and removing of applications are mentioned, as well as attaching and detaching platform devices. Application updates are checked by fetching XML encoded software update information from an Internet server. The *Web Ontology Language* (OWL) is used to describe platform devices and components with information, required for the self-configuration. The self-descriptions encoded in OWL are provided as knowledge for the *MAPE-K* loop.[12] A tackled self-configuration problem is the deployment of applications (resp. their components) to heterogeneous hardware. The Self-Configuration is performed by using constraint satisfaction problems (CSP). Two self-configuration algorithms are presented and compared by simulation, namely *backtracking* (worst-fit) and *Iterative repair* (min-conflict). The former algorithm is slower but better usable for building configurations from scratch, while the latter algorithm is faster, independent of the number of components and better usable if a configuration for the previous system state is given. The approach supports publish/subscribe and request/response communication. Graceful Degradation is mentioned in [99] as one aim of self-healing, ensuring that the most important functionality has to run with priority, but beyond that graceful degradation is not in key focus of their work. Also redundancy or replication mechanisms are not in their scope, instead they apply on-line task migrations as self-healing actions.

In [362], the authors present a self-adaptive framework to handle failures during runtime. As reaction to a failure, they determine a new valid deployment (allocation) of tasks to heterogeneous *Electronic Control Units* (ECUs). They consider algorithms that are aimed to be applied at runtime by the system

[11] Dynamically Self-Configuring Automotive Systems (DySCAS) [13], funded under FP6 from 2006–2008, `http://www.2020-horizon.com/DYSCAS-Dynamically-self-configuring-automotive-systems(DYSCAS)-s14987.html`

[12] MAPE-K: Monitor, Analyze, Plan, Execute with central self-knowledge [195]

itself. As a new valid deployment has to be found in a very short time to re-establish a valid system quickly after a failure, they focus on performance and scalability and look for a heuristic solution. In contrast to this, we focus on analyzing degradation scenarios of a system at its design time. If desired, our analysis results could be used to derive preconfigured runtime reconfigurations, avoiding to perform time-intensive recovery calculations at runtime, but limit the variation space of recovery actions. To support fault-tolerance for fail-operational features, we use redundancy mechanisms. Of course, this requires additional resources and is limited to failures caused by faults that can be anticipated at design time. They compare three heuristic optimization algorithms, *Simulated Annealing*, an *Evolutionary Algorithm* and *Tabu Search*, all based on Opt4J. Constraint solving by a SAT solver is also considered, based on SAT4J. The evaluation result was that SAT solving scales best. We use an SMT solver, which comprises SAT solving. They reach scalability with linear complexity for instance by considering *Earliest Deadline First* (EDF) scheduling instead of *Fixed Priority Preemptive* scheduling, what would force exponential complexity. They also tackle end-to-end timing analysis over the buses. We consider *Logical Execution Time* (LET) based cyclic scheduling, while the communication backbone ensures that each data item is available for all distributed subscribers in the next cycle. We both consider communication dependencies between software components resp. between tasks. We both do not focus on the detection of failures, but on how to handle detected failures. However, we treat degradations by explicitly deactivating components and also consider the effects on other components that depend on data items published by deactivated components. Furthermore, we handle mixed critical components and their separation into different clusters. Finally, we support degradations on feature level, combined with diversity. All three points are not considered by their approach.

One approach for dynamic reconfiguration in dependable systems is presented in [139], based on an architecture-level model and a runtime-level model. The intention is that dynamic reconfiguration in such systems needs to be causally connected at runtime to a corresponding high-level software architecture specification. Mixed criticality and fail-operational requirements are not considered. Graceful degradation is also not considered, instead they consider rollback mechanisms in case of failures.

Applications of AC or OC techniques to establish reliable fault-tolerant (respectively self-healing) embedded systems have been for instance introduced in [309], [5], [325], [65], [252], [324] and [134]. These approaches are intended to be applied at runtime by the system itself. The analysis introduced in this thesis is intended to be applied at design-time.

3.2 Constraint Based Synthesis of Design Decisions

In this section, we give an overview over problem solving approaches and frameworks, usable to synthesize optimized design decisions, most often as part of *design space exploration* (DSE) frameworks. We also show some applications, for instance to the deployment problem, which is also considered in this thesis.

The work in [62] states that constraint satisfaction problems (CSP, for instance scheduling or timetabling problems) that are looking for a feasible (any valid) solution are *NP-complete*, and that optimization problems looking for an optimal solution are *NP-hard*.

Example: Assembly Problem of Components: For instance, the assembly problem for component based systems is tackled in [235], by declaratively describing requirements, properties, interfaces, constraints (for replication, timing, separation) and objectives. This is a quite similar technique to our formal system model and our constraints and objectives. Merely, they solve the problem using a SAT solver, we use an SMT solver with optimization facility. We focus on a relatively problem specific synthesis and analysis approach for a specific class of system, while they consider a more generic view on the assembly problem of components, based in their interface descriptions, and propose a generic framework called CoBaSA (Component Based System Assembly). However, fail-operational features, failovers and degradations in failure scenarios are not tackled in their work.

Multi-Objective Design Space Exploration (MO-DSE): If an arbitrary valid solution for a problem is not sufficient, but a solution that is optimal in some sense is desired, then optimization objectives have to be expressed for the problem. If multiple competing objectives exist for an optimal solution, a design space is opened with pareto optimal solutions defining optimal trade-offs between the single objectives. The exploration of this design space is called design space exploration (DSE). Optimization problems with multiple conflicting objectives are called *multi-objective design space exploration (MO-DSE)* problems. Often appearing design questions are scheduling problems (task execution order) and deployment problems (task assignment to processors). These optimization problems are shown to have *NP-hard* complexity [333]. Certain multiple optimization objectives may exist, such as load balancing of the execution of tasks on execution units (processors) [329] [8], minimization of the number of required execution units (processors) for a given task set [254], minimization of missed real-time deadlines of tasks [254], or minimizing the maximum normalized response time of tasks [262]. Some of such objectives are conflicting, requiring to perform a trade-off analysis at the *Pareto front* of optimal combinations of solutions.

In [301], a generic DSE framework is introduced. They introduce an *Abstract Design Space Exploration Language* (ADSEL) as generic modeling framework for DSE problems, separating domain-specific from domain-independent DSE concepts. The abstract (domain-independent) language contains classes to model the a) design space, b) constraints, and c) objectives. The work includes a categorization of DSE problems into 1) resource allocation problems, 2) selection problems, 3) placement problems, 4) routing problems, 5) scheduling problems and 6) configuration problems.

1. **Resource allocation problems**: Mapping problem of resources to components, expressed in relational (many-to-many) or functional (one-to-one) manner. E. g., the problem of redundantly deploying software components to hardware resources, considered in this thesis, is a resource allocation problem. Due to the redundancy, we use a relational representation.
2. **Selection Problems**: Problem of finding a subset of a set, such that constraints are satisfied and an objective is optimized. E. g., in software product line engineering, the problem of selecting features to satisfy the constraints of a specific product in an optimal manner.
3. **Placement Problems**: Arranging objects according to geometric constraints, such that all objects lie within a given boundary and do not overlap.

4. **Routing Problems**: Find routes between nodes in a graph. E. g., in digital circuit design, wiring the pins of modules in a routing grid, while minimizing the length of wires.
5. **Scheduling Problems**: Calculate an activation order of tasks inclusive activation times for a schedule, such that temporal and resource constraints are satisfied.
6. **Configuration Problems**: Involves creation of a relationship between decision variables and their domain values subject to additional constraints.

In this thesis, we do not use a DSE language like ADSEL, but instead we use the API of an SMT solver to specify the model, the design space, the constraints and the objectives. We model the input problem in an XML file of a specific format, parse the XML file into our framework and convert the input problem model into an SMT model by using the API functions. In contrast to [301], focusing on a generic DSE framework, we focus on a specific DSE problem to support synthesis of optimal redundant deployments and analysis of degradation scenarios for a specific design type of fault-tolerant systems.

The work in [31] [30] introduces a model-driven multi-objective DSE (called *Octopus* tool set) of a low level micro-controller design, considering scheduling priorities and resource allocation alternatives (memory and page caching), and identifying optimal trade-offs between the required memory buffer size and the throughput of a microcontroller design. A so called DSE Intermediate Representation (DSEIR) is introduced to provide four views: application, platform, mapping, and experiment view. An analysis of timed automata is done using the Uppaal model checker.[13]

Other DSE applications are for instance the DSE of a generic hardware platform for software based products with objectives to minimize for instance the cost and the area of the hardware platform [229], a multi variant DSE for automotive E/E architecture hardware component platforms with robustness objectives supporting uncertain objectives and using Monte-Carlo Simulation [141] [142], an incremental semi-automatic DSE based on constraints defined as model transformation rules defined as *Prolog* predicates (applied to an automotive deployment example) [303], as well as the approaches shown in [363], [258], [215] and [282].

The work in [158] introduces a DSE approach incorporating guidance for the *selection* criteria and *cut-off* criteria. The aim of the cut-off criteria is to reduce the design-space by cutting off unpromising exploration states, while not cutting off an optimal solution. The result is a more efficient exploration.

Deployment of Software to Hardware: A lot of DSE approaches exist that focus on or support the problem of the deployment (alias allocation or mapping) of software to hardware. All of these have to be able to formulate constraints for valid deployments, as well as optimization objectives if an arbitrary valid deployment is not sufficient.

It is already mentioned for instance in [333] that the deployment optimization problem is a problem with *NP-hard* complexity. Hence, many approaches use heuristic approaches (see also section 2.7), not finding exact optimal solutions, but finding solutions close to an optimum in a relatively efficient time.

With focus on the domain of embedded real-time systems, the deployment (alias allocation or mapping) problem is for instance tackled by the following approaches:

- [167] tackles the allocation problem based on a constraint satisfaction problem (CSP). Deployment constraints are specified, for instance to take care that replications of components are distributed to multiple processors and not deployed to the same processor (due to fault-tolerance reasons). Also timing constraints for scheduling are defined. All constraints are weighted to make the cost of each one uniform in the objective function. They split the problem into a master problem handling the

[13] Uppaal: `http://www.uppaal.com`

deployment and resource constraints, and a subproblem for the timing problems. How the solver finds a solution is not in their focus.

- in [268], the authors introduce a design methodology for safety-critical systems, called SCRAPE (Safety-Critical Real-Time APlications Exploration), as well as the *fault-tolerant data flow* (FTDF) model of computation. They treat the efficient exploration of tradeoffs between *redundancy* and *cost*, considering time and space redundancy. The SCRAPE design flow has 6 main steps. Our work presented in this thesis is mainly related to steps 4 and 5, namely the specification of *Fault Behavior and Mapping Constraints* as well as the calculation of a *Fault-Tolerant Embedded Software Deployment*, including space redundancy. SCRAPE supports fail-silent execution platforms, what is similar to our assumption that failed execution units become isolated by appropriate platform mechanisms. However, we focus more explicitly on analyzing the fail-operationality of functional features during different degradation scenarios. Furthermore, we support to model and analyze degradations of single functional features (see section 4.7).

- in [55], fault-tolerant deployments with focus on the trade-off between performance and reliability are optimized using a *Mixed Integer Linear Programming* (MILP) solver. They also take replication of components into account. However, the approach does not consider mixed criticalities explicitly and only a single node failure model is supported. Degradation scenarios are not analyzed.

- in [26], task allocation in fault-tolerant distributed systems is tackled, using an approximation algorithm supporting the replication of tasks to processors. They consider several types of allocation (alias deployment) constraints, like separation constraints for replications, capacity constraints for processor memory, scheduling constraints for periodic real-time tasks, and load balancing constraints to improve reliability. The degree of approximation of an optimal allocation can be quantified depending on the amount of processors and the amount of required replicas per task. However, they do neither consider degradation scenarios in case of internal failures of system entities, nor they treat mixed-criticality (e. g., mixed DAL/SIL/ASIL, see section 2.1.5) or mixed-reliability (mixed fail-operational) requirements.

- in [364] a framework is introduced to optimize design of distributed embedded systems, particularly in the automotive domain. They consider software architectures consisting of tasks and messages, and priority based scheduling. They aim to optimize 1) the allocation (alias deployment) of tasks to electronic control units (ECUs), 2) the mapping of signals to messages, as well as 3) the assignment of priorities to tasks and messages in order to meet *end-to-end deadline* constraints and to minimize *latencies* of task finishing times and signal arrivals. To perform their optimization, they use a *mixed integer linear programming* (MILP) framework. To reduce the complexity, they divide their optimization into an approximating two-step synthesis. Step one is about synthesizing task allocations based on a heuristic about task and signal priorities. Step two is about synthesizing the mapping of signals to messages, as well as the final priorities of tasks and messages. However, opposed to the approach introduced in this thesis, they do not consider redundancies or replications in allocations (alias deployments) to ensure mixed fail-operational requirements, improving reliability. They also do not consider degradation scenarios resulting from internal failures of system entities. Moreover, in this thesis we not only synthesize deployments for various scenarios incorporating an adequate level of redundancy to meet fail-operational requirements, but we also synthesize the communication channels between software components out of a set of channel candidates, appearing from publish/subscribe definitions of communication ports of components, enabling us to synthesize deployments that cause minimal network traffic.

- in [365], the authors introduce the two metrics *cohesion* and *coupling* to measure the quantitative quality of a safety relevant deployment. The *cohesion* metric describes the increase of cost to develop

a software component with a higher *Safety Integrity Level* (SIL) than normally required, because this component is desired to run in the same partition as another component with a higher SIL. The *coupling* metric describes the costs introduced by the safety critical communication mechanisms. These metrics enable the description of optimal deployments with respect to safety-related costs. As further constraints that can be used to limit the solution space, the authors mention target platform type and dissimilarity of ASWCs allowing to define that the target types of a set of ASWCs have to be disjoint. The authors propose a *Genetic Algorithm* (GA) to solve the deployment problem. However, the authors mention that there was no special reason for using a GA, but only because the work was embedded into an infrastructure where a GA was already used. The authors claim that also other techniques, like *Linear Programming* (LP), might be used. SAT or SMT solvers are not mentioned.

- Multi-objective optimization of the deployment of components to automotive networks is shown [156]. Redundancy is not tackled, but mentioned as necessary future work. Graceful degradation is also not considered.

- in [210], optimal designs of automotive architectures are tackled by introducing a generic multi-objective evolutionary optimization framework. A *Domain Specific Language* (DSL) named *Automotive Architecture Optimization Language* (AAOL) is introduced allowing to express constraints, objectives and orders, for instance for deployment and scheduling problems. The orders are used to create ranks for the multi-criteria optimization. The approach does not explicitly focus on redundancy mechanisms, failure scenarios, fail-operational requirements, failovers and degradations, like we do in this thesis. However, the DSL may be usable or extensible to express the model, constraints, objectives and optimization orders introduced in this thesis, whereupon one of the different solving and optimization strategies could be used to create solutions as basis for our analysis. However, in this thesis we just apply an SMT solver and use its API to create instances of our formal model containing the input model of the system under analysis. We parse the output of the SMT solver to obtain its solutions and extract those parts that represent the result of our analysis.

  In [211], the above approach is more generalized to a DSL named *System Architecture Optimization Language* (SAOL) and applied to an example from the avionics domain. Earliest-Deadline-First (EDF) is considered as scheduling policy. Different problem solving and optimization strategies are incorporated and can be used to solve the problem formulated in SOAL, like MOEA, SMT and ILP-based approaches. Also a guided improvement algorithm is supported.

- [228] [227] provides modeling, analysis, and optimization of automotive networks, focusing on efficient symbolic multi-objective design space exploration. They mention that multi-objective evolutionary algorithms (MOEA) are good in many DSE problems, but might fail in design spaces that contain only a few feasible solutions. Contrary to MOEA, they propose a multi-objective 0-1 ILP solving approach, incorporating a heuristic based on pseudo-boolean (PB) solvers, as well as a backtracking algorithm to tackle multi-objective problems. During their DSE, they assume two models as input, namely 1) a hardware resource architecture including processors and buses, as well as 2) a software application architecture including tasks and directed data-dependencies between the tasks. Based on this, they treat three design spaces during the DSE: 1) decision about which resources shall be used in the final system (called *allocation*), 2) decision about which task becomes executed on which processor resource (called *binding*), 3) decision about the *routing* of messages on the bus resources (only in [227]). The *binding* is that what we consider as deployment in this thesis. Other than we do it in this thesis, they do not explicitly consider constraints for valid redundancy or replication mechanisms, mixed fail-operational requirements, as well as the analysis of failure scenarios and resulting failover and degradation scenarios. We apply an SMT solver, as our focus is not on providing a most efficient solving technology.

- in [361], a comparison is shown about deployment calculation by a SAT solver and by the *simulated annealing* algorithm. The result was that SAT solving scales better and is more efficient for larger sets of equations. The use case of the work is to find a new valid software allocation (alias deployment) in case of a component failure. This means, the allocation calculation has to be performed as fast as possible to ensure a quick healing of the system.
- [354] introduces an iterative DSE approach to enable the user to influence the prioritization of contradicting multi-objectives. As example, they consider the trade-off between 1) *cost* (in €), 2) *weight* (in g) and 3) the amount of electronic control units (ECUs) for an automotive lane-keeping support system. The objective is to minimize all three criteria. They express timing constraints by using the *Timed Augmented Description Language v2* (TADL2)[14], supporting for instance the specification of end-to-end deadlines, for instance for the event chain from video sensing to electric power steering (EPS). They also consider the deployment (alias allocation) of tasks to clusters (partitions) of execution units and buses. Their DSE cannot optimize multiple objectives at a time, hence they obtain optimized results for each optimization objective independently. Additional DSE runs are performed subsequently incorporating user prioritization for the objectives.
- a DSE for deployment and scheduling synthesis for mixed-criticality shared memory multicore architectures is presented in [347], based on using the YICES SMT solver [107]. The DSE is integrated [345] within the model-based AUTOFOCUS3 (AF3) software and systems engineering tool [205] [341] [15].[15] Also a DSE of deployments towards the PikeOS separation kernel operating system [190] has been introduced in [305] to address separation of concerns for mixed-criticality applications in an ISO26262 [174] context. The AF3 DSE framework is usable quite flexibly, for instance to ensure ASIL-conformant deployments for safety critical automotive systems [348] [368]. Multi-Objective DSE in AF3 is based on the Z3 SMT solver [93], using the integrated optimization facilities of Z3 [52] to provide Pareto efficient solutions. Currently, also a domain specific modeling language to specify constraints and objectives is under development [110].
- in [271], the definition of a heterogeneous hardware platform is addressed by introducing different viewpoints for the stakeholders. Mixed critical hierarchical software components get deployed to the described hardware platform in a *Software Allocation Planning View*. Existing deployment algorithms are reused. The described deployment constraints only contain limitations of software components to get deployed to a specified subset of the hardware nodes. Redundant deployments, failure scenarios and degradation scenarios are not considered. In [270], the *Object Constraint Language* (OCL) [16] is used to define deployment constraints and multiple objectives. Afterwards, a transformation is done from OCL into a boolean 0-1-ILP problem, and a LP-Solver is used to calculate the solution.
- a DSE approach for UML/MARTE models [255] is introduced in [164]. The DSE is based on simulations and is performed in iterative loops. They distinguish so called DSE *rules* and DSE *constraints*, both modeled as UML comments decorated with MARTE stereotypes and additional stereotypes of the introduced COMPLEX toolkit. The DSE rules are compliant with what we call constraints in this thesis, for instance constraints for valid allocations of software to hardware, reducing the exploration space during their simulations. The DSE constraints that they introduce refer to performance constraints. However, these DSE constraints to not prevent the evaluation of any points in the design space during their simulation. The topics at which we focus on in this thesis, namely adequate redundancy in deployments, analysis of failure scenarios and resulting degradation and failover scenarios, are not tackled in their work.

[14] TADL2 was developed in the TIMMO-2-USE project, `https://itea3.org/project/timmo-2-use.html`
[15] `http://af3.fortiss.org`
[16] `http://www.omg.org/spec/OCL`

- [357] uses a *Genetic Algorithm* (GA) to solve the optimization problems for UML/MARTE [17] and EAST-ADL2 [18] models for AUTOSAR systems. The provided framework is called *Architecture Framework for Modeling Analysis and Optimization* (AFfMAO). As the MARTE meta-model does not include sufficient support to specify optimization objectives, they introduce a so called *Generic Optimization Modeling* (GOM) profile to model objectives. The aim of the work is *computer aided configuration*, for instance for the deployment of software components to ECUs, as well as the mapping of AUTOSAR runnables to AUTOSAR tasks. To reduce the runtime of the GA, they propose a heuristic two-step deployment approach, combining a divide-and-conquer strategy with an iterative improvement. The divide-and-conquer strategy divides the deployment problem into two sub-problems. They distinguish different types of optimization strategies for *time-driven* and *data-driven* task activation models of AUTOSAR. However, in contrast to this thesis, redundancy, fail-operational requirements and degradations of system functionality in failure scenarios are not considered in their work.

- [153] introduces an iterative design space exploration (DSE) approach with focus on system robustness and performance, based on a multi-objective evolutionary algorithm (MOEA). As the DSE focuses on *robustness*, we discussed this work already in section 3.1.2.

- a DSE approach for reliability-driven optimization of deployments in embedded systems is presented in [238] [237], performing the DSE based on evolutionary algorithms. We discussed this work already in section 3.1.2.

- [272] [273] considers software deployment for distributed embedded real-time systems of automotive applications, focusing on the configuration of the communication infrastructure and how to handle design constraints.

- in [213], hierarchical communicating components are deployed using a *Hierarchical Genetic Algorithm* (HGA). The subcomponents get instantiated with different levels of redundancy, in order to maximize the system reliability. However, graceful degradation in failure scenarios is not considered.

- in [165], the authors show a graph-based approach to deploy communicating software components to a distributed system. The objectives are to minimize the cost and to maximize the reliability of a deployment. The cost is either measured in the number of deployed components, or in the number of required target hosts. Reliability is considered with respect to network failures. To increase reliability, the deployment is optimized to maximize local communication and minimize remote communication. This also increases performance. However, redundancy and degradation of system functionality in failure scenarios are not considered.

The deployment problem is tackled in many more approaches: [232] uses a multi-objective genetic algorithm (MO-GA) to solve the problem of mapping a set of task graphs onto a heterogeneous multiprocessor platform, also considering the scheduling problem; [333] uses simulated annealing (SA) to allocate hard real-time tasks to processors. Further work considering the deployment problem is for instance introduced in [234], [16], [327], and [9].

[17] Modeling and Analysis of Real-Time Embedded Systems, `http://www.omg.org/omgmarte`

[18] Electronics Architecture and Software Technology-ADL, `http://www.east-adl.info`

Scheduling: The synthesis of optimized real-time schedules (task execution orders on processors, and/or message transmission over network) is tackled for instance in [111] and [347]. The latter approach performs a MO-DSE, combined with the optimization of deployments of software components to hardware execution units. A DSE for schedule synthesis of multi-period (multi-rate) software components is presented in [346]. However, none of these approaches consider explicitly failure scenarios and resulting degradations of the set of system features.

CHAPTER 4

Analyzing Fail-Operational and Fail-Degrading Systems

In this chapter, we introduce the contribution of this thesis. As described in section 1.2, the motivation of this thesis is to introduce an approach to formally analyze failure scenarios and to ensure the fulfillment of fail-operational requirements in mixed criticality and mixed reliability embedded systems, after isolations of assumed failing software or hardware components. We consider the deployment of software to hardware as an open design space, and synthesize valid deployments for each failure scenario, ensuring the fulfillment of all fail-operational requirements by incorporating needed degradations and failovers, if feasible.

Contents

4.1 Introduction to the Formal System Model 66
4.2 Formal System Model 70
4.3 Concept Overview 83
4.4 Properties of System Model Elements 84
4.5 Synthesis of Valid Redundant Deployments 95
4.6 Analysis of Failure Effects 98
4.7 Supporting Degradations of Single Functional Features 122
4.8 Formalization of Optimization Objectives 140
4.9 Assumptions and Aspects that are out of Scope 142
4.10 Explanations and Discussions about the Formal System Model 144

Outline: In section 4.1, we provide an informal introduction about the problem domain, representing the kind of systems that we aim to describe and analyze with the introduced approach. In section 4.2, we introduce the formal model that we use to describe the system under analysis and its relevant properties as basis for the analysis that we introduce. Section 4.3 gives an introduction into the analysis procedure, used model artifacts and used tools. The specified input problem properties and variable solution properties are introduced in section 4.4. Section 4.5 shows how we synthesize valid redundant deployments based on formalizations of constraints for valid deployments. An initial consideration of failure effect analysis, based on the introduced formal model and formal failover and degradation constraints, is presented in section 4.6. In section 4.7, the presented approach is extended by the analysis of degradations on functional feature level, which is an important ability especially for systems requiring diversity and functional degradation. A formalization of optimization objectives is presented in section 4.8. Assumptions for our approach are discussed in section 4.9, as well as design and analysis aspects that are out of scope of this thesis. Three examples are embedded into the sections illustrating to which kinds of input problems our approach is applicable and which kind of degradation and failover scenarios can be analyzed. We conclude this chapter with explanations about the design decisions for the formal model, as well as discussion of alternatives, in section 4.10.

4.1 Introduction to the Formal System Model

In order to tackle our research questions, the formal system model has to be designed in a way that allows to express properties of system elements, as well as constraints with respect to valid deployments of software to hardware and valid failover and degradation scenarios. We start to introduce the formal model by first introducing a common concept of viewpoints in system design, based on which we introduce the formal model later on.

4.1.1 Viewpoints

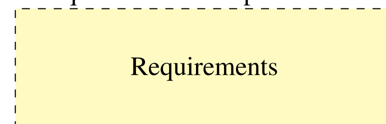
When designing and modeling a system, different viewpoints have to be considered. For instance, in the project SPES XT [1] four viewpoints are used, called 1) Requirements Viewpoint, 2) Functional Viewpoint, 3) Logical Viewpoint and 4) Technical Viewpoint [269]. The four viewpoints are introduced and distinguished in [269] as follows:

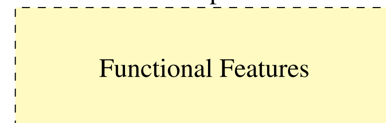
Requirements Viewpoint: eliciting, documenting, negotiating, and validating requirements for the system under development. Different types of requirements, like assumptions, constraints, goals, behavioral or technical requirements are distinguished in four types of models.

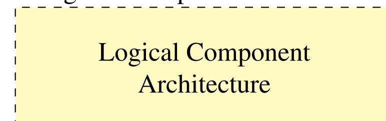
Functional Viewpoint: development of a functional system specification for the system under development. Functional requirements become traced to user functions, and user functions become specified by system functions. Functional dependencies and feature interactions can be analyzed [342].

Logical Viewpoint: solution design for the system under development. Functions become realized by communicating logical components, structured with respect to an architectural design. In this design, aspects like dependability, maintainability or reusability are important.

Technical Viewpoint: technical implementation of the system under development. The logical components become refined to technical software components (SWCs), which become deployed and executed on a hardware execution platform, comprising hardware execution units, operating systems, etc.







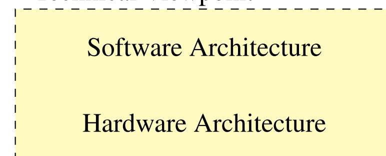


Figure 4.1: SPES Viewpoints

One focus of the work in [269] lies on seamless integration of the different viewpoints. Each viewpoint can be considered in different abstraction layers. However, in this thesis we do not distinguish

[1] Software Platform Embedded Systems (SPES) XT, `http://spes2020.informatik.tu-muenchen.de/spes_xt-home.html`

different layers of abstraction. The introduced formal system model shall be used to represent the system under analysis with an adequate level of abstraction containing all pieces of information needed to perform the introduced analysis.

Adherence of formal system model to the SPES viewpoints: The formal system model, which we introduce below in section 4.2, is spread over the functional viewpoint and in the technical viewpoint, and allows to trace between these.

We do not focus on the requirements viewpoint, as we do not model the functional requirements that led to the creation of the elements of the functional viewpoint. Instead, we treat the elements of the functional viewpoint as given for the system under analysis. The formal system model allows to describe a non-hierarchical set of *functional features*, representing the functional viewpoint.

However, we express informal requirements for valid deployments, valid failover mechanisms and valid degradations by formal constraints and formal optimization objectives, expressed over the elements of the technical viewpoint.

For the sake of simplicity, we do not model the logical viewpoint. In [269], the logical components of the logical component architecture (modeled in the logical viewpoint) become mapped in a n:m manner to software components of the software architecture, modeled in the technical viewpoint. However, in this thesis we assume an 1:1 mapping between logical components and technical software components. Due to this, we do not incorporate the logical viewpoint into our formal system model. Instead, we model a direct tracing between functional features (of functional viewpoint) and technical software components (of technical viewpoint). The technical software components become deployed to the execution units of the hardware architecture. This deployment is part of the technical viewpoint.

Summarized, we consider as part of the formal system model:

- Functional Viewpoint:
  - A set of *functional features* that the system under analysis shall provide to fulfill its functional requirements.
- Technical Viewpoint:
  - *Software Architecture* comprising software components (SWCs), realizing functional features by software. Software components may have communication ports and are connected by communication channels to be able to interact.
  - *Hardware Architecture* comprising physical hardware execution units

We enrich our model elements with formal properties, partially representing requirements, like fail-operational requirements for functional features. These requirement properties are part of the input model for our analysis, representing the system under analysis. We model formal constraints cross over the viewpoints as representation of requirements for valid system design, like for instance for valid initial redundant deployments and valid reconfigurations of the deployments in failure scenarios.

4.1.2 Meta-Model of Considered System Structure

Fig. 4.2 introduces a meta-model in Unified Modeling Language (UML) class diagram notation about the problem domain that we consider. The meta-model describes the structure of the system under analysis that we support with the introduced approach. However, it is not completely equivalent to the formal system model that we introduce later, but the formal system model represents certain parts of the meta-model in a refined mathematical fashion. We show the meta-model here as introduction to the problem domain of the supported types of systems.

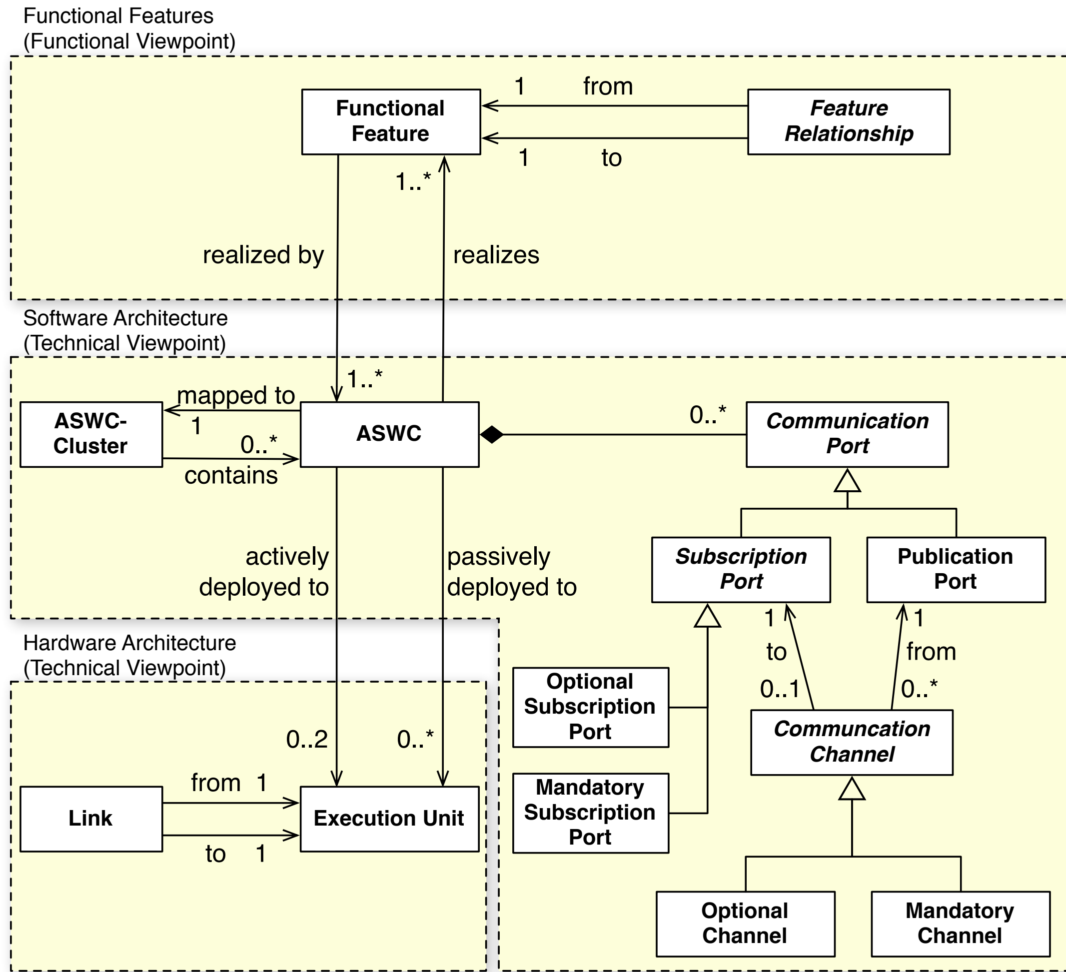


Figure 4.2: UML meta-model of the considered problem domain for the system under analysis

The meta-model contains functional features and application software components (ASWCs) that realize (implement) these features. A functional feature is realized by one or more ASWCs, while an ASWC can contribute to realize one or more features, resulting in a *n:m* relationship.

The ASWCs become deployed onto execution units, like electronic control units (ECUs) in the automotive domain. We distinguish active and passive deployments. Passively deployed ASWCs are stored in memory, but not executed. The deployment can be done in a limited redundant manner. In case of a redundant deployment, multiple instances of an ASWC exist, deployed to different execution units. At most two active instances may exist at the same time (one master and one hot-standby slave). However, in degradation scenarios, it may happen that no active instance exists anymore. This means that the realized functional feature cannot be provided anymore and becomes unavailable.

We also treat communication channels between ASWCs. We model communication dependencies between ASWCs with the publish/subscribe paradigm and distinguish mandatory and optional communication by mandatory and optionally subscribed data. An ASWC can operate with missing optional input data, but cannot operate with missing mandatory input data.

Finally, during the analysis, certain execution units and/or software components are assumed to have failures and to get isolated from the residual system by appropriate RTE fault-tolerance mechanisms. We describe a formal approach to analyze how the system has to be degraded in order to meet the reliability requirements of all functional features in such failure scenarios.

The model supports a grouping mechanism of application software components into so called ASWC-Clusters. The consideration of clusters is inspired by the clustering concept investigated in the RACE project, see section 2.5.2. The clusters contain ASWCs with identical safety and reliability requirements.

We need the formal model as basis to be able to formally express *constraints* for valid deployments, valid degradations and valid failover mechanisms. Some of the constraints can be expressed in an *arithmetic* manner, using for instance sums over the values of properties of system elements and comparison operators like "the sum of the required memory of the software components deployed to an execution unit must not exceed the provided memory of that execution unit". Other constraints can be expressed in a *logical* manner, using for instance implications like "if a software component X is lost, then functional feature Y cannot be provided anymore". Often, both kinds of expressions have to be combined to formalize a constraint. Finally, the formal model deals as input for an out of the box problem solving and optimization technology, which calculates solutions of variable properties in the valid constrained solution space of the formal model. We set up the model and the variable properties of model elements such that it becomes possible to analyze degradation scenarios and failover scenarios based on the calculated solution.

4.1.3 Motivation and Benefits of the Formal System Model

In the next section 4.2, we define the formal system model that we use as basis for the synthesis and analysis approach which we introduce in this thesis. We define the formal model to establish a foundation on which we can setup several steps that are required for the aimed analysis of degradation and failover scenarios in case of failures of system elements. The formal model, which we are going to introduce in the subsequent part of this chapter, is designed to enable to express the following pieces of information and to be usable as input for an SMT solver.

The major motivation and design objectives for the construction of the formal system model are to formally express:

- *constraints* for
  - valid (partially redundant) deployments of software to hardware,
  - valid failover scenarios of redundant instances of software components,
  - valid degradation scenarios of the set of available functional features.
  We model all constraints in form of combinations of arithmetic and logical expressions (e. g., conjunctions, disjunctions, implications). We do not use any uninterpreted functions or quantifiers in the implementation of the formal constraints. Instead of quantifiers, we roll out all constraints in a for-loop over the model elements.

- a *solution space* for the synthesis of a valid initial deployment, as well as deployments for degradation and failover scenarios caused by assumed failures of hardware execution units and/or software components

- *optimization objectives* for initial and degraded deployments, in form of arithmetic maximization or minimization expressions over the model

Further secondary design objectives for the formal model are to formally express:

- requirements of functional features, like fail-operational requirements
- properties of software components, like worst case execution times
- interface ports of software components, as well as communication channels between these ports
- properties of hardware execution units, like provided time budget to execute software components
- tracing between functional features and the software components that realize the features

4.2 Formal System Model

In this section, we describe a formal system model that we constructed with the aim to allow to express formal constraints for valid deployments, degradation-scenarios and failover-scenarios over this model and to create instances of this model, representing the system under analysis as input for the analysis approach introduced in this thesis.

Overview of the major elements of the formal system model: In Section 4.1, Fig. 4.2, we introduced an UML meta-model representation of the problem domain that we tackle in this thesis. The formal system model - that we use as basis for the analysis approach introduced in this thesis - is partially equivalent with the problem domain meta-model, but however some information is represented in a different way and some information is omitted because it is not required to perform the analysis. Fig. 4.3 shows a class-diagram representation of the formal system model. In contrast to the meta-model of the problem domain (Fig. 4.2), all inheritance and all abstract classes (like *Communication Port*) are removed, because the formal system model does not employ inheritance or abstract classes.

Furthermore, we model communication channels, deployment relationships and mappings of application software components (ASWCs) to ASWC-Clusters as properties of a *System Configuration* element. The *System Configuration* contains system wide information, as well as derived information and design decisions that we synthesize as part of the presented approach. The design decisions are conducted during our analysis in order to determine a valid system design that fulfills the requirements of functional features, in particular requirements to be fail-operational. For instance, the *System Configuration* contains matrices storing the selected communication channels between ASWCs (*CM* and *CO*), the determined mappings of ASWCs to ASWC-Clusters (*map*), as well as the calculated valid deployment of ASWCs to execution units (*deploy*).

We omit the *Feature Relationships* in the formal system model, because we do not need to model them to perform our analysis. Instead, we analyze their representation in the software architecture in form of communication channels between ASWCs. Despite communication channels are usually seen as part of the software architecture, we model the channel matrices *CM* and *CO* as part of the *System Configuration*, as the selection of channels is part of our synthesis approach. Realizations of functional features by ASWCs are represented by χ and χ^{-1}, introduced in more detail later in section 4.2.3.

We introduce the different shown elements of Fig. 4.3 subsequently in the below sections and refine them by adding additional properties to the elements, required to express pieces of information needed to perform our analysis.

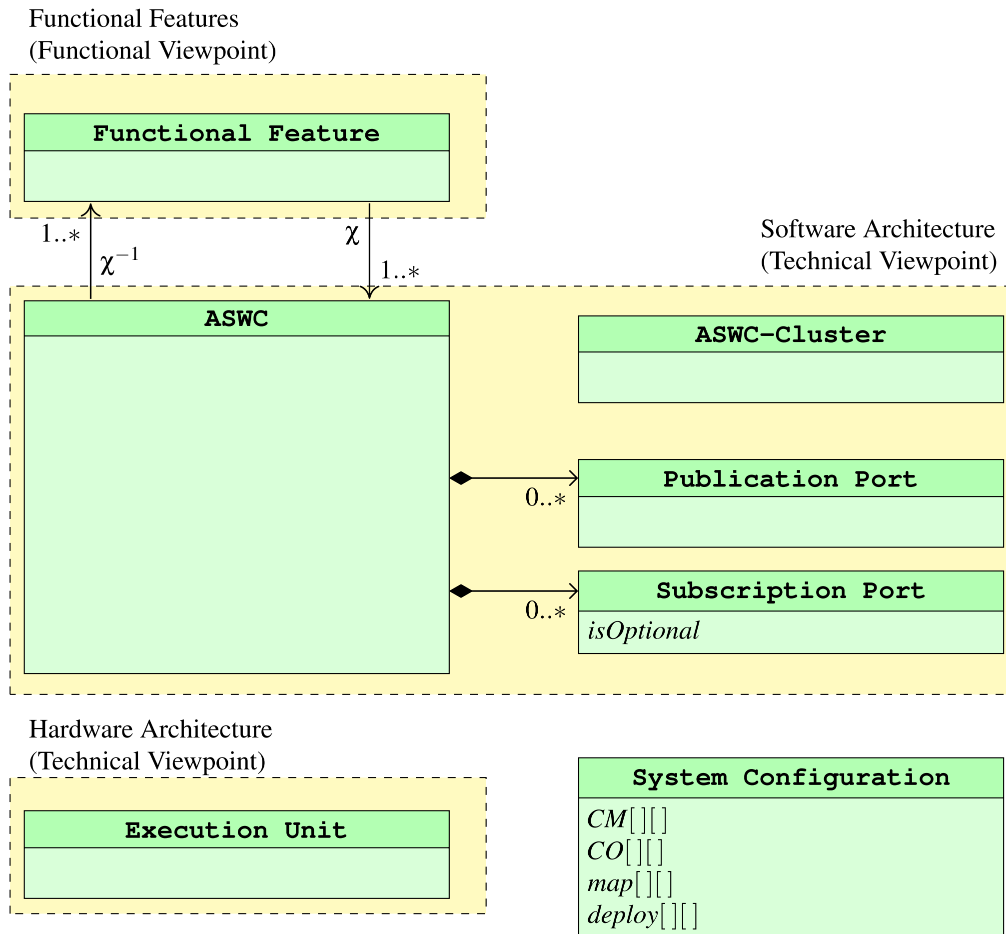


Figure 4.3: Class diagram representation of the formal system model

Relationship between formal system model elements and entities of the really existing or designed system: The introduced formal system model contains abstract identifiers of entities of the real system under analysis.

Particularly, as shown in Fig. 4.4, the model contains:

- a set F of identifiers of functional features that are provided by the system under analysis
- a set S of identifiers of application software components, designed and implemented for the system under analysis
- a set E of identifiers of hardware execution units, incorporated in the system under analysis

When we talk later on for instance about functional features of set F, we always implicitly mean the identifiers which are the elements of set F. During the presented analysis, we assign properties to the elements of the sets. We also obtain analysis results, which we assign as properties to the elements of the sets. Although the properties are assigned to the set elements being identifiers, we can deduce that these properties hold for the corresponding system entities, as there is a direct bijective mapping between

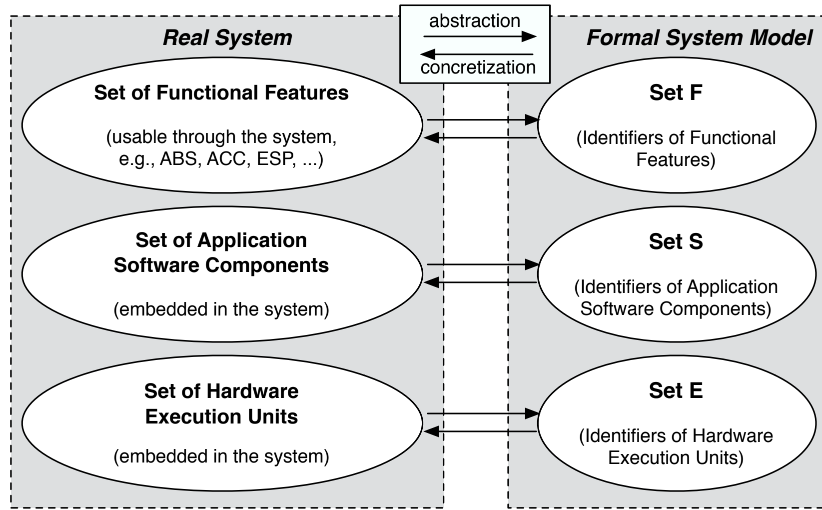


Figure 4.4: Bijective mapping between sets of entities of real system and sets of identifiers of formal system model

real world system entities and their identifiers in the formal system model. If a certain analysis result is attached in the formal model for instance to an identifier of a functional feature in form of a property value, we deduce that this property value holds for the unambiguously related real world functional feature.

Formal notations: In sections 4.2.1 - 4.2.6, we formally define all the system elements that are sketched in Fig. 4.3. In the formal definitions, we denote the power set over a set X by $\mathcal{P}(X)$, which is the set of all subsets of set X, inclusive the empty set $\emptyset$ and X itself. We write $\mathcal{P}^+(X)$ for the power set of set X without the empty set, meaning that $\mathcal{P}^+(X) = \mathcal{P}(X) \setminus \emptyset$. In addition, we use the cartesian product of sets $A \times B = \{(a,b) \mid a \in A, b \in B\}$ to define a matrix over the elements of sets A and B. As usual, we use $\mathbb{N}_0$ for the infinite set of natural numbers inclusive zero (equal to the non-negative integers). We use $\mathbb{N}^+ = \mathbb{N}_0 \setminus \{0\}$ for positive natural numbers without zero.

We are now going to introduce the elements of the formal system model.

4.2.1 Functional Features

We now introduce those elements of our formal system model, which represent the functional viewpoint.

Definition 1 – Functional Features: We define F as a finite set of identifiers of functional features, and $f \in F$ as a single identifier of a functional feature.

This means, the finite set $F = \{f_1, ..., f_m\}$, with $m \in \mathbb{N}^+$, contains identifiers of the functional features that the system under analysis shall provide. We assign additional properties to the functional features, introduced later in section 4.4.

We do not consider compositional feature hierarchies in this thesis. We discuss the relation between feature set F and feature hierarchies in section 4.10.1. We do not model the behavior of functional features, as the aim of our analysis is to check on structural level if a feature can be provided at all in an analyzed degradation scenario, not if the behavior of that feature is correct or incorrect.

4.2.2 Software Architecture

Software architectures are typically based on a software *component model*, being a specification of component types, allowed patterns of interactions among instances of these components types, and between components and a component runtime environment [166]. Different component models are for instance discussed in [220]. In our formal model, we assume a single type of component, and we describe the *interfaces* of software components by the use of communication *ports*. A port is a logical point of interaction between a software component and its environment [319]. We assume a component model distinguishing two types of ports: publication ports and subscription ports. A publication port is a port that provides and asynchronously sends data. A subscription port is a port that requests and asynchronously receives data. We assume that the components of the assumed component model are executed in a layered architecture as part of an application layer, being on top of a runtime environment (RTE) layer, as it is the case for instance in AUTOSAR (see section 2.4.2) and RACE (see section 2.5.1).

We now introduce a formal model of the elements of the assumed component model.

Definition 2 – Publication ports: We define $\mathtt{PP}$ as a finite set of identifiers of publication ports, and $pp \in \mathtt{PP}$ as a single identifier of a publication port.

Definition 3 – Subscription ports: We define $\mathtt{PS}$ as a finite set of identifiers of subscription ports, and $ps \in \mathtt{PS}$ as a single identifier of a subscription port.

For all publication ports and subscription ports, we specify the sent or received data by an identifier for communicated data, named data-ID. We model the data-ID as a property of the ports, introduced in section 4.4.2. To be able to connect a publication port with a subscription port, the data-IDs of the both ports must be identical. The data-IDs can also be used to encode publish/subscribe communication design based on *topics*, like for instance applied in DDS [259] and ROS [278], or to abstract from the combination of *topics* and *attributes* applied in XME [71]. We do not consider subtyping of data in this thesis for simplicity.

Software Components

We use the introduced publication and subscription ports to specify interfaces of software components. A *software component* (SWC) is a self-contained piece of software with dedicated interfaces, which can individually be deployed to and executed on a piece of hardware execution unit [328].

To distinguish software components located at different layers of a layered architecture like AUTOSAR or RACE, in this thesis we use the term *application software component* (ASWC) to address software components located at the application layer of the software architecture of a system. ASWCs realize (alias implement) the functional features F of a system.

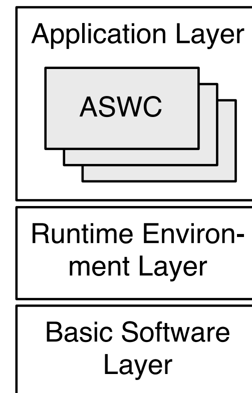


Figure 4.5: ASWCs in a layered software architecture

Definition 4 – Application Software Components (ASWCs): We define S as a finite set of identifiers of application software components (ASWCs), and $s \in S$ as a single identifier of an ASWC.

Hence, the finite set $S = \{s_1, ..., s_n\}$, with $n \in \mathbb{N}^+$, contains identifiers of the application software components (ASWCs) that are designed for the system under analysis.

Each ASWC is composed of a finite set of publication ports and a finite set of subscription ports.

Definition 5 – Ports of ASWCs: To each ASWC $s_i \in S$, with $i \in \mathbb{N}^+$ being an index to distinguish different ASWCs, we associate

- a finite set $\mathtt{PP}_i = \{pp_{i,1}, ..., pp_{i,y}\}$ of publication ports and
- a finite set $\mathtt{PS}_i = \{ps_{i,1}, ..., ps_{i,z}\}$ of subscription ports.

Both sets can be empty. Hence, we declare $y, z \in \mathbb{N}_0$. To address the union of all publication and subscription ports of s_i, we introduce the set $\mathtt{P}_i = \mathtt{PP}_i \cup \mathtt{PS}_i$.

We write $pp_{i,k} \in \mathtt{PP}_i$ to address the kth publication port of ASWC s_i and $ps_{i,k} \in \mathtt{PS}_i$ to address the kth subscription port of ASWC s_i, with $k \in \mathbb{N}^+$. We use $\mathtt{PP}_i$ as a short notation for $\mathtt{PP}(s_i)$, and $\mathtt{PS}_i$ as a short notation for $\mathtt{PS}(s_i)$.

Software Component Example: Fig. 4.6 shows an example for an ASWC $s_1 \in S$, having three subscription ports $\mathtt{PS}_1 = \{ps_{1,1}, ps_{1,2}, ps_{1,3}\}$ and two publication ports $\mathtt{PP}_1 = \{pp_{1,1}, pp_{1,2}\}$.

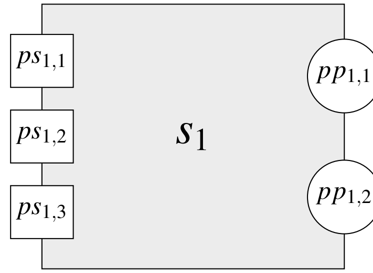


Figure 4.6: Example ASWC with 3 subscription ports and 2 publication ports

Communication Channels

Based on the ports of ASWCs, *directed data-flow communication channels* are established between the ports, transferring data messages from publication ports towards one or multiple subscription ports. In case of the used publish/subscribe paradigm, the communication between ports is performed asynchronously [116] by the RTE of the system under analysis, coordinated for instance by a *Broker* in the RTE [71].

Furthermore, we distinguish mandatory and optional subscription ports, under the assumption that ASWCs have some input data which is mandatory to provide their intended service, and that ASWCs have other input data which is only optional to provide their intended service. We distinguish the two kinds of subscription ports by a property *isOptional* : $\mathtt{PS}_i \rightarrow \{0, 1\}$, being introduced in more detail in section 4.4.2.

However, the amount of ports might become quite huge and hence, also the space of possible communication channels. To reduce the size of the formal system model, we do not directly incorporate channels between ports of ASWCs. Instead, we only model the aggregated information about channels between ASWCs, because this is sufficient for the analysis that is introduced in this thesis. We use two matrices for storing optional and mandatory communication channels between the ASWCs, as defined below.

Definition 6 – Matrices of Directed Communication Channels: We define $CM : S \times S \to \mathbb{N}_0$ to be a matrix representing *mandatory* directed data-flow communication channels between ASWCs. We define $CO : S \times S \to \mathbb{N}_0$ to be a matrix representing *optional* directed data-flow communication channels between ASWCs.

For a publisher ASWC $s_i \in S$ and a subscriber ASWC $s_k \in S$, matrix cell $CM(s_i, s_k)$ stores information about the directed communication channels from the publication ports $\mathtt{PP}_i$ of s_i towards the *mandatory* subscription ports $\{ps_{k,l} \in \mathtt{PS}_k \mid isOptional(ps_{k,l}) = 0\}$ of s_k. Each matrix cell $CO(s_i, s_k)$ does the same for channels towards *optional* subscription ports $\{ps_{k,l} \in \mathtt{PS}_k \mid isOptional(ps_{k,l}) = 1\}$ of s_k.

If the software architecture of the system under analysis contains multiple channels from publication ports $\mathtt{PP}_i$ to subscription ports $\mathtt{PS}_k$ of the two ASWCs $s_i, s_k \in S$, we aggregate these channels in our model to a single mandatory and a single optional channel between the respective ASWCs. This is sufficient for the aims of our analysis and reduces the size of the model, allowing more efficient calculations of solutions and hence, more efficient analysis. Although we consider channels between ASWCs, we define ports of ASWCs in the input problem model to describe possible communication channel candidates between ASWCs. However, based on the given ASWC-Ports from the input problem model, we select used channels out of these candidates as part of our analysis, enabling the synthesis of software architectures and deployments that engender a low network traffic. We discuss this in more detail in section 4.10.2.

Channel Example: Fig. 4.7 shows an example of the channel matrices CM and CO between three ASWCs. Publication ports of ASWCs are drawn as circles, subscription ports are drawn as squares. The values of the channel matrix cells are left blank with dots initially here. We introduce the values subsequently.

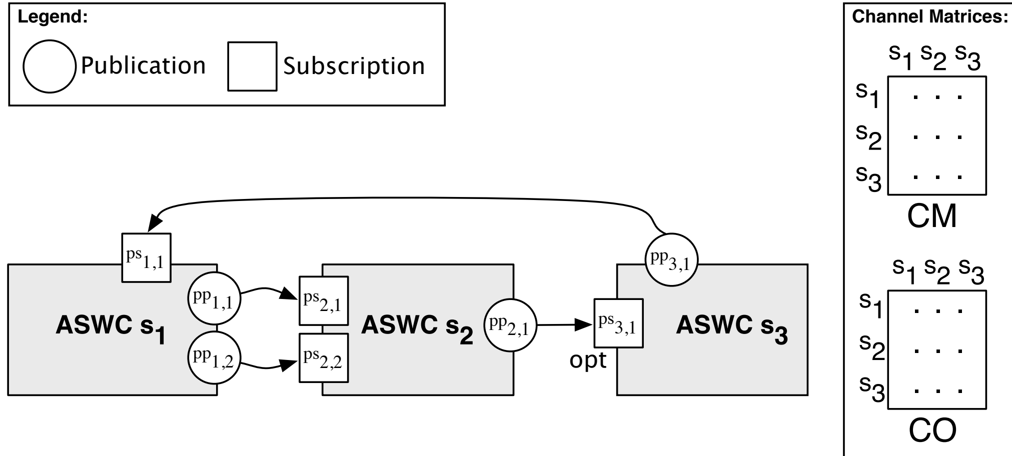


Figure 4.7: Example of channels between ASWCs

In the example shown in Fig. 4.7, ASWC s_1 has two publication ports $pp_{1,1}$ and $pp_{1,2}$, ASWC s_2 has one publication port $pp_{2,1}$ and s_3 has one publication port $pp_{3,1}$. Similarly, the subscription ports $ps_{k,l}$ can be seen in the figure, as well as the channels that are synthesized to connect the ports. Notice that the publication and subscription port indexes are numbered independently from each other, as a port can never be publisher and subscriber simultaneously. There exist three mandatory subscription ports $ps_{1,1}$, $ps_{2,1}$ and $ps_{2,2}$, as well as one optional subscription port $ps_{3,1}$ (labeled with 'opt'). Mandatory and optional subscription ports share their index space. However, in this example no ASWC has both mandatory and optional subscriptions.

Weights of Channels: The value of an entry of a channel matrix denotes the *aggregated weight* ω of all directed channels that exist from publication ports $\mathtt{PP}_i$ of $s_i \in S$ towards mandatory respectively optional subscription ports $\mathtt{PS}_k$ of $s_k \in S$. A weight ω represents the amount of data that is transfered over a channel in a period of time, e. g., per execution cycle. Publication ports have a weight $\omega : \mathtt{PP}_i \rightarrow \mathbb{N}^+$ denoting the amount of published data. The value of a matrix cell $CM(s_i, s_k)$ resp. $CO(s_i, s_k)$ for $s_i, s_k \in S$ are the aggregated weights ω of the subset of publication ports of s_i that are connected to subscription ports of s_k. We use the weights during the deployment synthesis to prefer local communication, and by this to reduce network traffic. We select channels out of a set of channel candidates and aggregate them to the introduced channel matrices.

We introduce the weights of ports in more detail in section 4.4.2. The values of the elements of the channel matrices we introduce in more detail in section 4.4.3, Eq. 4.2. We describe the channel selection mechanism in more detail in section 4.4.3, Fig. 4.15.

Fig. 4.8 enriches Fig. 4.7 with weights $\omega(pp_{i,j})$ of publication ports $pp_{i,j} \in \mathtt{PP}_i$ of $s_i \in S$. Based on these weights, the channel matrices are filled with values.

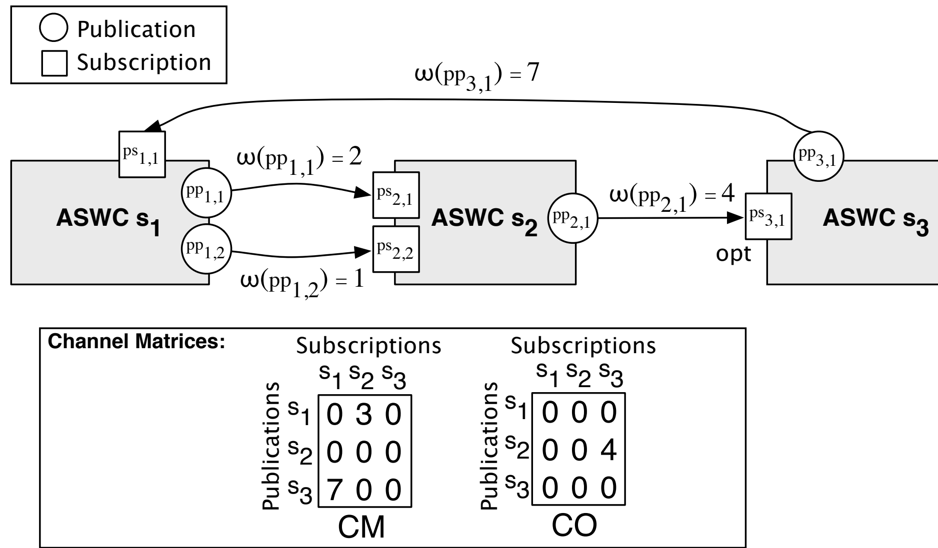


Figure 4.8: Example of channels between ASWCs inclusive weights

The weights of the publications ports in the example are: $\omega(pp_{1,1}) = 2$, $\omega(pp_{1,2}) = 1$, $\omega(pp_{2,1}) = 4$ and $\omega(pp_{3,1}) = 7$. The bottom part of Fig. 4.8 shows the aggregated channel matrices CM and CO. The upper left corner of CM denotes the entry $CM(s_1, s_1)$, the lower right corner the entry $CM(s_3, s_3)$. Same for CO. The channel creation is based on data-IDs that are assigned to the ports (introduced in section 4.4.2). We assume in this example that the shown channels are valid. There exist two mandatory channels from s_1 to s_2 with weights $\omega(pp_{1,1}) = 2$ and $\omega(pp_{1,2}) = 1$. In CM, the weights of these both channels are aggregated to $CM(s_1, s_2) = 3$. There exist one optional channel in the example from s_2 to s_3, causing $CO(s_2, s_3) = 4$.

Clusters of ASWCs

The analysis approach introduced in this thesis supports the consideration of groups of ASWCs. We call these groups *ASWC-Clusters*, as introduced below.

Definition 7 – ASWC-Cluster: We define C as finite set of identifiers of ASWC-Clusters, and $c \in C$ as a single identifier of an ASWC-Cluster.

The finite set $C = \{c_1, ..., c_q\}$, with $q \in \mathbb{N}^+$, contains ASWC-Clusters, which are groups of ASWCs. The ASWC-Clusters are structure building elements.

We use the clusters as motivated in section 2.5.2 to group ASWCs with identical requirements for safety (ASIL) and reliability (fail-operational by redundancy), preparing the separation of mixed critical and mixed reliable ASWCs from each other, as different clusters can be separated using spatial and temporal partitioning mechanisms [292].

Each ASWC $s \in S$ shall be mapped to exactly one ASWC-Cluster $c \in C$. We model this mapping as defined below.

Definition 8 – ASWC-Cluster Mapping: We define $map : S \times C \to \{0,1\}$ to be a matrix that represents the mapping of ASWCs S to ASWC-Clusters C. We define the elements of map as:

$$map(s,c) = \begin{cases} 1 : s \in S \text{ is mapped to } c \in C \\ 0 : \text{otherwise} \end{cases} \tag{4.1}$$

We synthesize a valid mapping as part of our approach and ensure the validity of the obtained mapping by formal constraints about the introduced formal system model. We introduce in section 4.5.1 how we express formal constraints. As we assume that the mapping is not predefined in the input model, but we obtain a valid mapping as part of our analysis, we store the mapping as part of the system configuration Φ, see section 4.2.5.

The mapping of ASWCs to the ASWC-Clusters is a total function, but not necessarily injective or surjective, as multiple ASWCs can be mapped to the same cluster and clusters might be empty in the formal model.

Cluster Example: Fig. 4.9 shows an example for a mapping map of three ASWCs onto two ASWC-Clusters.

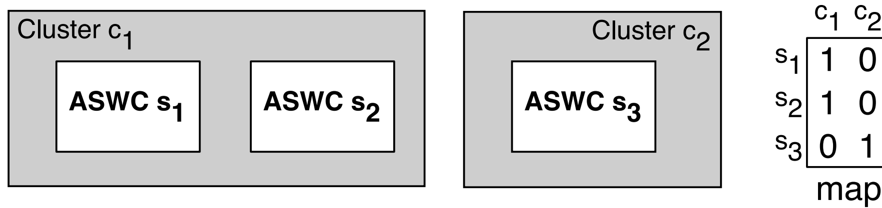


Figure 4.9: Example of a mapping of ASWCs to ASWC-Clusters

The application software components are composed and connected by the communication channels in order to realize the feature set F of the system. The result of this composition is a software architecture. In a valid software architecture, each mandatory subscription port has to be connected to a publication port, to be able to execute the ASWC to which the subscription port belongs to. We ensure this by formal constraints. If a mandatory subscription port has no valid matching publication port, the analysis is canceled. In such a case, either the solver returns that the model is unsatisfiable, or by using the soft-constraints introduced later in section 4.6.7 the solver returns a satisfiable model including an information that the corresponding constraint cannot be satisfied.

4.2.3 Feature Realization

When constructing the application software components of a system, it is decided in which way the functional features of set F become realized by software components of set S.

We introduce a *feature realization* relationship between a certain functional feature and the subset of ASWCs that realize this feature. Each feature is realized by one or more ASWCs, while each ASWC contributes to realize one or more features. We formally model this relationship between functional features and ASWCs as follows.

Definition 9 – Feature Realization: Let χ denote a feature realization mapping. We define χ and its inverse χ^{-1} as:

$\chi : F \to \mathcal{P}^+(S)$ with $\chi(f) = \{s \in S \mid s \text{ contributes to realize } f \in F\}$

$\chi^{-1} : S \to \mathcal{P}^+(F)$ with $\chi^{-1}(s) = \{f \in F \mid f \text{ is partially or completely realized by } s \in S\}$

Hence, χ associates a functional feature $f \in F$ to those ASWCs that realize f by software. We neglect in this thesis the physical sensors and actuators and further mechanical, hydraulical or electrical hardware components, which may also be required to realize a functional feature (like inverter of electric motor or brake piston). We focus on the software component part of the feature realizations. The software components interact with the physical world by sensor and actuator interfaces.

The feature realization mapping χ is given as part of the input model for our analysis. We model this given mapping as formal constraint using *implications*, and use it to express constrains like "if $s \in \chi(f)$ is lost or has to be explicitly deactivated due to insufficient resources, then f cannot be kept available".

Feature Realization Example: Fig. 4.10 shows an example for Def. 9. Two features $F = \{f_1, f_2\}$ exist. Feature f_1 is realized by three ASWCs s_1, s_2 and s_3, hence $\chi(f_1) = \{s_1, s_2, s_3\}$. Feature f_2 is realized only by s_3, hence $\chi(f_2) = \{s_3\}$. ASWC s_3 contributes to realize both features, hence $\chi^{-1}(s_3) = \{f_1, f_2\}$.

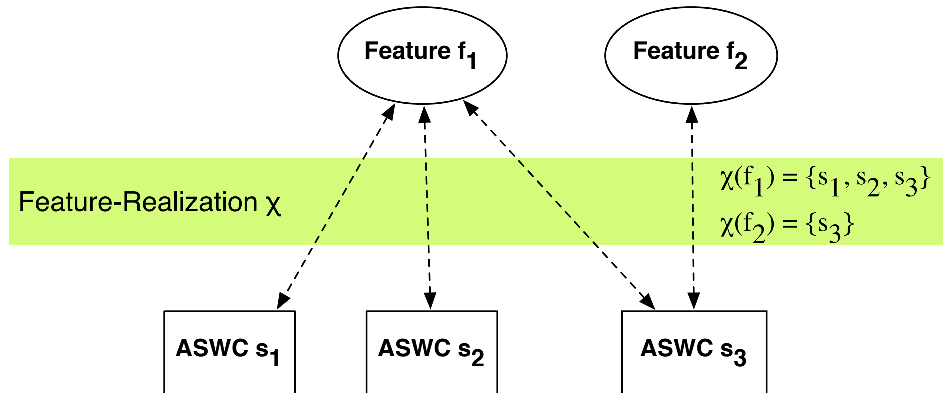


Figure 4.10: Example of realization relationship between features and ASWCs

4.2.4 Hardware Architecture

Definition 10 – Execution Units: We define E as a finite set of identifiers of hardware execution units, and $e \in E$ as a single identifier of an execution unit.

The execution units are also called Electronic Control Units (ECUs) in the automotive domain, or *Duplex Control Computers* (DCCs) in the RACE approach. We assume that the execution units are interconnected by a reliable physical communication topology, ensuring a reliable communication between all units. Hence, we do not consider the physical communication topology furthermore in this thesis.

4.2.5 System Configuration

The system configuration contains certain information that is derived during our analysis from the input model, as well as design decisions that are conducted as part of the synthesis during our analysis. In particular, this concerns the mapping of ASWCs to ASWC-Clusters, the communication channels between ASWCs, as well as the deployment of ASWCs to execution units. We synthesize this configuration as part of our analysis and do not assume a predefined configuration as part of the input model, as the freedom on these design decisions denotes a huge added value for our analysis. Instead of just rejecting an invalid predefined configuration that violates for instance fail-operational requirements of some functional features or violates constraints for valid redundant deployments, we can synthesize a valid configuration that fulfills the requirements, if there exists such a valid configuration.

As major part of the system configuration, we determine a valid deployment of software components to execution units, incorporating adequate redundancy to meet the fail-operational requirements of the associated realized functional features. We store the result in the deployment matrix defined below.

Definition 11 – Deployment: We define $deploy : S \times E \rightarrow \{0, P, M, HS\}$ to be a matrix representing the deployment of ASWCs S to execution units E. For $s \in S$ and $e \in E$, we define the elements $deploy(s,e)$ as:

$$deploy(s,e) = \begin{cases} 0 : s \in S \text{ is not deployed to } e \in E \\ P : s \in S \text{ is deployed to } e \in E \text{ } \mathbf{p}\text{assively (cold-standby slave)} \\ M : s \in S \text{ is deployed to } e \in E \text{ actively as } \boldsymbol{master} \\ HS : s \in S \text{ is deployed to } e \in E \text{ actively as } \mathbf{h}\text{ot-standby } \boldsymbol{slave} \end{cases}$$

We choose this encoding as we need to distinguish the three types of deployment to cover the different types of considered redundancy, introduced in section 2.5.2. We distinguish passive deployments and two types of active deployments. In a *passive* deployment, the binaries of ASWCs are in memory of execution units, but not executed in schedule. In an active deployment, the ASWC binaries are instantiated and executed in schedule. Active *master* instances provide the functionality that realizes the functional features. The redundant *hot-standby slave* instances are actively in schedule, but the published data items are ignored by the RTE of the system. Hence, hot-standby slaves do not contribute to realize functional features, but in case the master instance is lost due to a failure, the hot-standby slave can be transformed to become the new master. This transformation is called a *failover*. The benefit of having a hot-standby slave, compared to having only a passive backup (cold-standby slave), is that the internal state of the hot-standby slave is kept synchronous with the master state and hence, in case of a failover the behavior can be continued without any intermediate cold startup behavior from initial state. See also Fig. 2.6 in section 2.1.6 for distinction between hot- and cold-standby slaves. See section 4.10.5 for further discussions about matrix *deploy*. When implementing the formal model as arithmetic input model for the SMT solver, we use the following encoding for the symbolic values: P=1, M=2, HS=3.

Deployment Example: Fig. 4.11 shows an example for a deployment of ASWCs to execution units. The ASWCs s_1 and s_2 are deployed as master without redundancy to execution unit e_1. As s_1 and s_2 are in the same cluster c_1, they are always deployed to the same execution unit. The third ASWC s_3 is deployed redundantly to both execution units, once actively as master to e_2 and once passively (as cold-standby slave) to e_1. In this example, no hot-standby slaves exist. Hence, matrix *deploy* shown in Fig. 4.11 does not contain any entry with value HS.

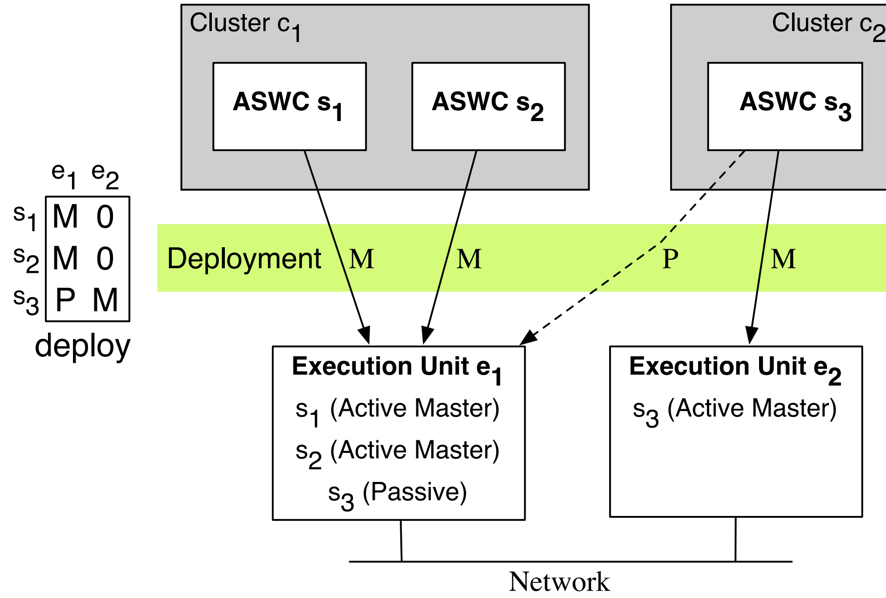


Figure 4.11: Example of a deployment of ASWCs to execution units

Finally, a system configuration Φ contains all properties that are conducted as design decisions as part of our synthesis and analysis. In particular, these are those parts of the solution of our analysis that cannot be attached to a single instance of a model element, but correlate to relationships between multiple model elements.

Definition 12 – System Configuration: Let Φ denote a system configuration. We compose the system configuration of $\Phi = \{CM, CO, map, deploy\}$.

The system configuration contains the solution properties that store the synthesis of a valid configuration of communication channels *CM* and *CO*, the mapping *map* of ASWCs to ASWC-Clusters, as well as the deployment *deploy* of ASWCs to execution units (see also Fig. 4.3).

4.2.6 System Model

Finally, we aggregate all introduced formal model elements to compose a formal system model.

Definition 13 – System Model: Let $\mathbb{S}$ denote a formal system model. We compose the formal system model of $\mathbb{S} = \{F, S, C, E, \Phi, \chi\}$.

4.2.7 Example for the Formal Definitions

Fig. 4.12 shows an integrated example for the definitions of the formal system model. A feature set $F = \{f_1, f_2\}$ is realized by ASWC set $S = \{s_1, s_2, s_3\}$, as specified by feature realization χ. The ASWCs are deployed to two execution units $E = \{e_1, e_2\}$. ASWCs s_1 and s_2 are mapped to the same cluster $c_1 \in C$ and are deployed without redundancy to execution unit e_1. The third ASWC s_3 is mapped to another cluster $c_2 \in C$ and is deployed redundantly to both execution units, once actively as master to e_2 and once passively to e_1.

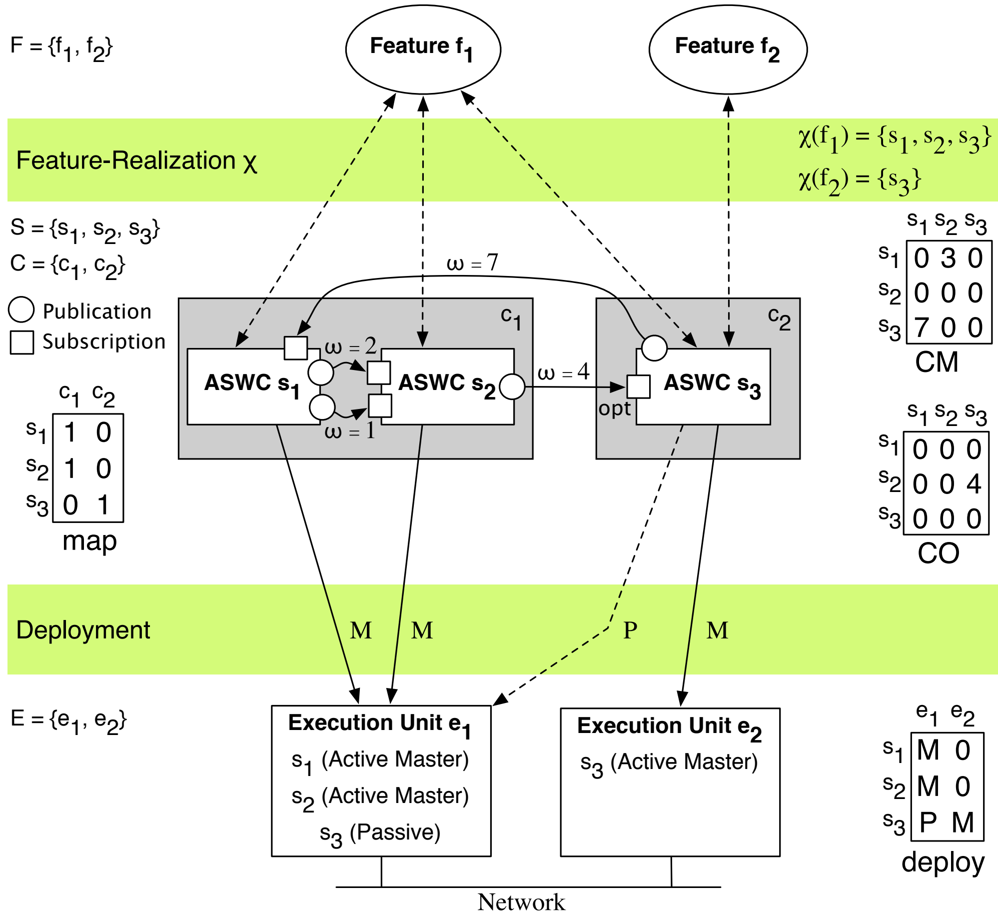


Figure 4.12: Example for the formal system model

The example contains publication ports and subscription ports of ASWCs and the resulting communication channels. Communication channels with a total wight of 11 units flow over the physical network between the two shown execution units e_1 and e_2, as ASWC s_3 is active as master on e_2 and receives data from s_2 and sends data to s_1, while s_1 and s_2 are active as master on e_1. Hence, $CO(s_2, s_3) = 4$ and $CM(s_3, s_1) = 7$ have to be transmitted over network. $CM(s_1, s_2) = 3$ is a local communication inside e_1.

However, in failure scenarios the amount of data transmitted over the network might change. For instance, in case of a failure of execution unit e_2, a failover has to be performed to let s_3 on e_1 become the new master. The remote communication becomes a local communication.

Furthermore, we assume here that the three ASWCs cannot be executed on the same execution unit due to insufficient resources. This means, in the above sketched failover scenario, we assume that s_2 has to be deactivated on e_1 to be able to activate the passive backup of s_3 on e_1. As s_3 has only an optional input from s_2, this is valid. If s_1 would be deactivated, also s_2 has to be deactivated as s_2 receives mandatory data from s_1, and s_1 has no redundancy. However, because s_2 has to be deactivated to activate s_3 on e_1, the feature f_1 cannot be kept available anymore. Our analysis approach would reveal this as part of the analysis of degradation scenarios for different cases of failures of execution units or ASWCs. If feature f_1 would have a requirement to be fail-operational after one failure of an execution unit, the analysis reveals that this requirement cannot be fulfilled with the shown system design.

4.2.8 Summary Overview of Formal Model Symbols

In table 4.1, we give a summary of the so far introduced symbols representing elements of our formal system model.

Table 4.1: List of Formal Model Symbols

Symbol	Explanation
F	Set of Functional Features
f	One Functional Feature
S	Set of Application Software Components (ASWCs)
s	One Application Software Component (ASWC)
χ	Subset of ASWCs that contribute to realize a given functional feature $f \in F$
χ^{-1}	Subset of functional features to whose realization a given ASWC $s \in S$ contributes to
$\mathtt{PS}_i$	Set of Subscription Ports of ASWC s_i
$ps_{i,j}$	j^{th} subscription port of ASWC s_i
$\mathtt{PP}_i$	Set of Publication Ports of ASWC s_i
$pp_{i,j}$	j^{th} publication port of ASWC s_i
$\omega(pp_{i,j})$	weight of port $pp_{i,j}$, describing the amount of published data
CM	Matrix of communication channels from publication ports of one source ASWC towards mandatory subscription ports of one destination ASWC
CO	Matrix of communication channels from publication ports of one source ASWC towards optional subscription ports of one destination ASWC
C	Set of ASWC Clusters
c	One ASWC Cluster
E	Set of Execution Units
e	One Execution Unit
Φ	System Configuration
$\mathbb{S}$	Aggregated System Model

4.3 Concept Overview

Before going into further details about additional input properties and solution properties of the elements of the introduced formal model, in Fig. 4.13 we show a brief view onto the tooling architecture and procedure of the introduced analysis, to provide a better understanding about the purpose of the properties and how we use them.

The input model, describing the system under analysis, is described in a XML file (A) and parsed (B) into the input part of the formal model (C) by adding constraints to set the input properties to fixed specific values, given in the XML file. The input model includes the set of functional features F, the set of software components S, the ports $\mathtt{PP}_i$ and $\mathtt{PS}_i$ of a component $s_i \in S$, the realization relationships χ, the set of execution units E, as well as fixed input model properties of these elements (being introduced in section 4.4.2). During parsing the input XML file into the formal model, additional constraints are added (D), for instance to represent the relationship $\chi(f)$ between a functional feature $f \in F$ and the ASWCs which realize f.

In addition to these input problem specific properties and constraints, we also add generic solution properties, constraints and objectives. The set of solution properties represents the open decision variables of the solver (E). These solution properties are introduced in sections 4.4.3 and 4.4.4 and contain the results of our analysis. We add generic constraints which model valid initial redundant deployments, valid degradation and failover scenarios, and other constraints required to perform the analysis (F). We introduce these constraints subsequently in sections 4.5.1, 4.6.5, 4.6.6 and 4.7.4. Optimization objectives specify in which way the solver shall calculate the values of the solution properties (G). The objectives are discussed in section 4.8.

We implemented our framework in Python, using the Python API of the Z3 SMT solver to interact with it.

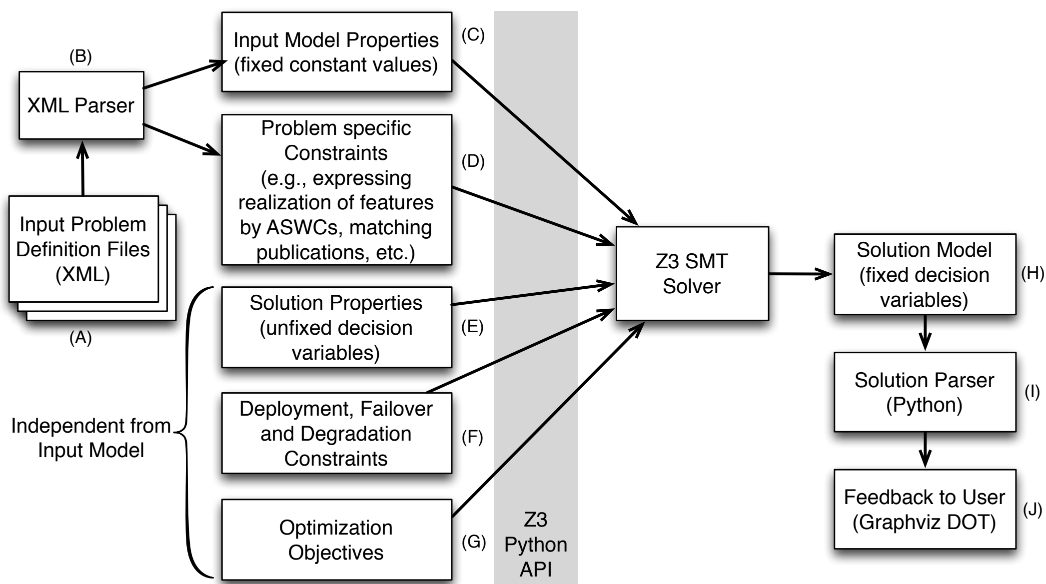


Figure 4.13: Brief visualization of the tooling and the procedure how we use the introduced formal model

After all properties, constraints and objectives have been added to the solver model, the solver function to calculate the solution model is executed and an optimized solution model is returned, if existing (H). We parse the solution (I) and represent the analysis result to the user in form of a visualization of the deployments and the availability of features in the different degradation/failover scenarios, rendered using the Graphviz *DOT* language (J). [2]

It is also possible to execute the solver multiple times, obtaining multiple points on the pareto frontier, representing pareto efficient solutions for a multi-objective problem with contradicting objectives, requiring trade-offs. Alternatively to the implementation in Python, the framework can be also implemented in C/C++ or Java, as Z3 offers also APIs to these languages.

The introduced analysis approach enables the user to formally analyze a system design with respect to its fulfillment of fail-operational requirements on a structural level, as well as to analyze the level of degradation that is needed in different failure scenarios. This is combined with the synthesis of valid deployments of software components to execution units, automatically incorporating an adequate level of redundancy and failovers to meet the fail-operational requirements.

4.4 Properties of System Model Elements

In this section, we introduce the elements of our formal system model in more detail. We enrich the elements of the formal system model with properties that describe the elements. We use the term *property* here similarly to the definition of a *component property* in [166]:

> *Component Property*: Something that is known and detectable about a component [166]

In [166] they use a dot notation *component.property* to represent properties of components, and use the types of *string*, *float*, *integer* and *boolean* to specify the possible values of a property. However, we do not apply the dot notation in this thesis. Based on the so far introduced formal system model, we model the properties as follows.

Definition 14 – Property: We define a *property* to be an identifier for a function $foo : domain \rightarrow codomain$, having a *domain* of one of the introduced formal system elements, and a defined *codomain*.

For instance, if we would like to assign a property named *prop* to the ASWCs of the set *S*, assigning to each $s \in S$ a natural number $\mathbb{N}$, we would define the property as $prop : S \rightarrow \mathbb{N}$.

In the next section 4.4.1, we provide an overview over all the properties that we are going to introduce in more detail subsequently in sections 4.4.2, 4.4.3 and 4.4.4. We need these properties to represent the system under analysis adequately, to be able to perform the calculation of valid deployments and the degradation and failover analysis in failure scenarios. The properties represent both the input model of the system under analysis (see section 4.4.2), as well as the results of our analysis (see sections 4.4.3 and 4.4.4).

[2] `www.graphviz.org/content/dot-language`

4.4.1 Overview of Properties of Formal System Model

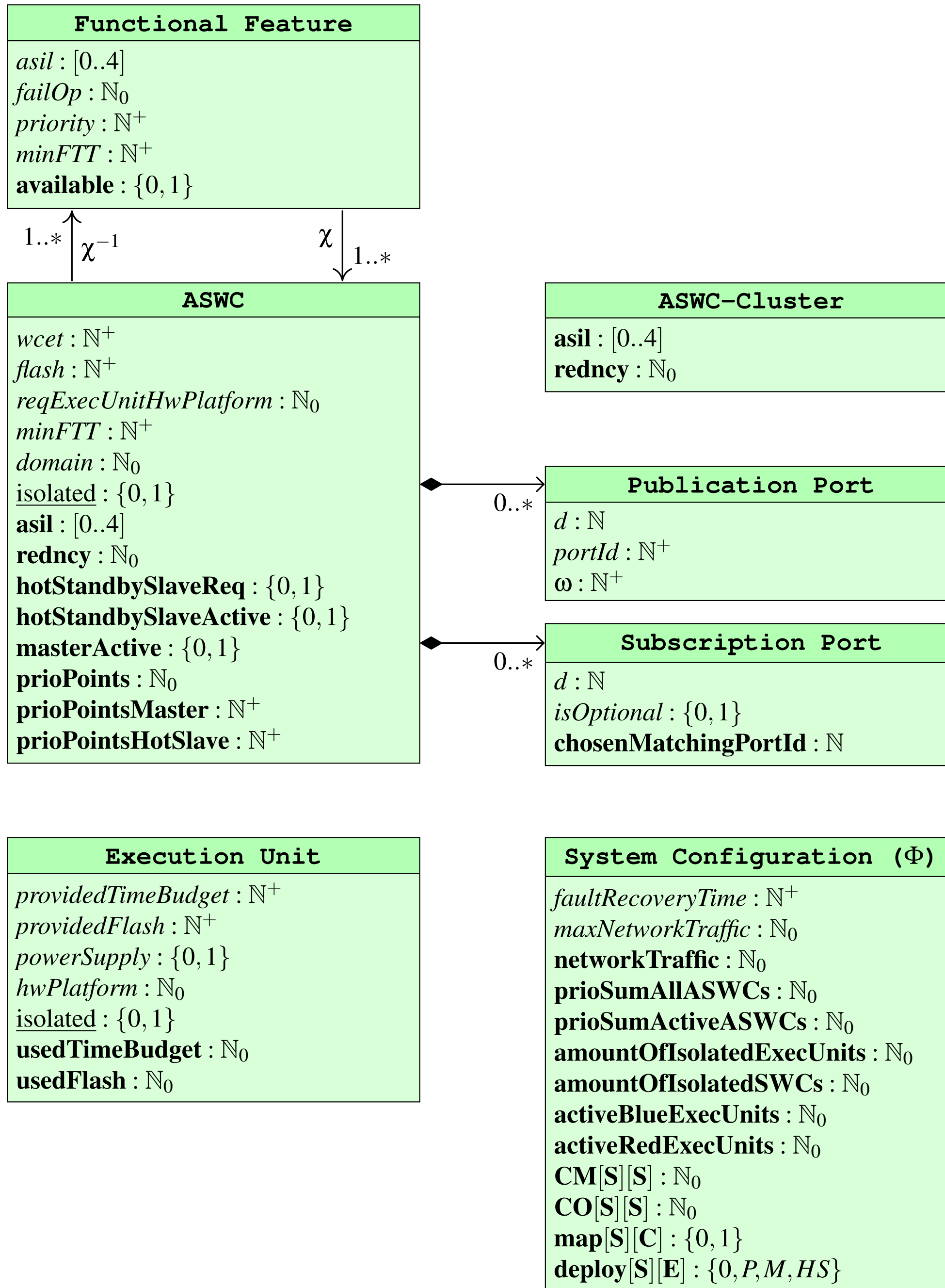


Figure 4.14: Class-diagram representation of the formal model, incl. properties as class attributes

Fig. 4.14 shows an extended version of the class diagram representation of the formal system model that was shown in Fig. 4.3. The figure is enriched by all the properties of model elements, which become introduced in sections 4.4.2, 4.4.3 and 4.4.4. The properties that are drawn in *italic* font are the input problem properties, fixed to a certain value and describing the system that is desired to be analyzed (the *system under analysis*). The properties drawn in **bold** font are the solution properties, representing the decision variables of the solver and finally containing the result of the analysis. The underlined properties are neither input nor solution properties, but properties that are variated during our analysis to different values to represent the different failure scenarios.

For instance, the *asil* property is only given on feature level in the input model. The **asil** property of ASWCs and ASWC-Clusters is then derived in the solution model from the related features according to χ^{-1} and the *map* property. The isolated flags of ASWCs and execution units are the properties controlling the failure scenarios, being analyzed for necessary degradations or failovers to fulfill all fail-operational requirements of functional features.

System Configuration properties: We enrich the system configuration Φ by additional properties, representing pieces of information on system level. System level information can be:

1. System level *requirements* or predefined system level *design decisions*: We model system level requirements as properties inside the input model for our analysis, having a fixed value (e. g., *maxNetworkTraffic*). The same holds for predefined system level design decisions. The requirements and predefined decisions are used in constraints over the formal model, influencing the validity of deployment solutions and the validity of degradation and failover scenarios.

2. System level properties representing design decisions conducted by our synthesis approach, or properties *deriving* from these synthesized design decisions: Those system level properties that represent or derive from the synthesized valid deployment and degradation/failover scenarios are modeled as variable properties (shown in **bold** in Fig. 4.14, e. g., **networkTraffic**), belonging to the solution part of the formal model.

4.4.2 Input Model Properties

In this section, we describe those properties of the formal model, which represent the input problem for the deployment calculation and failure effect analysis. Failure effects may be required degradations, or failovers needed to ensure fail-operational requirements. The input properties are fixed a priori to certain values, describing the feature set, the set of software components with their ports, the set of hardware execution units, as well as some system wide configuration properties of the system under analysis.

System Configuration Φ: We assume that there is only exactly one system configuration. We define a property $\mathit{faultRecoveryTime} : \mathbb{N}^+$, which is the *fault-recovery time* (FRT) of the system (in ms). The *fault-recovery time* is a predefined system property, stating how quick the system, particularly the systems RTE, is able to detect an error or a failure of a system element, isolate this element from the residual system, and perform a failover to a redundant spare element to recover from the causing fault. This means, the RTE guarantees that the service or data that is provided by the failing system element is unavailable at most for the fault-recovery time, if the corresponding functional feature is required to be fail-operational. For instance, the RACE RTE is designed to guarantee this (c.f. section 2.5.2). The *fault-recovery time* property influences the type of redundancy (cold- vs. hot-spares) that has to be used in the deployment.

In addition, property $\mathit{maxNetworkTraffic} : \mathbb{N}_0$ defines a required restriction for the maximum amount of network traffic, arising from communication channels between ASWC instances that are deployed to different execution units.

Functional Features F: Property $asil : F \rightarrow \{0..4\}$ defines the *Automotive Safety Integrity Level* (ASIL) of a feature [0: Quality-Management (QM), 1: ASIL A, 2: ASIL B, 3: ASIL C, 4: ASIL D]. Property $failOp : F \rightarrow \mathbb{N}_0$ defines in which sense the feature is required to be fail-operational. If $failOp(f) > 0$, feature $f \in F$ must be kept available after the first $failOp(f)$ failures of any one execution unit or software component. In addition, features have a $priority : F \rightarrow \mathbb{N}^+$ that can be used to specify user preference about which features are desired to be kept available longer than others. The higher the value of $priority(f)$, the higher is the intended priority of feature $f \in F$ and the longer this feature will be kept available. However, the priority is only used as tie-breaker between features with identical fail-operational level. Hence, property *priority* can not be used to specify that a non-fail-operational feature should be kept available, while a fail-operational feature becomes unavailable. Finally, each functional feature has a property $minFTT : F \rightarrow \mathbb{N}^+$, which denotes the minimum *fault-tolerance time* (in ms) of that feature for its different safety goals (see section 2.5.2). The *minFTT* describes how long a failure of a functional feature can be tolerated temporarily and hence, how quickly associated realizing slave software components have to be enabled to become new masters to recover the feature.

Application Software Components S: Property $wcet : S \rightarrow \mathbb{N}^+$ defines the *Worst-Case Execution Time* of a unique cyclic executable operation of each ASWC (in μs). Property $flash : S \rightarrow \mathbb{N}^+$ defines the required amount of flash memory to store the binary of an ASWC (in kilobyte). Property $reqExecUnitHwPlatform : S \rightarrow \mathbb{N}_0$ defines the hardware platform of the execution units, to which instances of an ASWC are allowed to be deployed to, in case ASWCs are not hardware independent (see also property *hwPlatform* of execution units, introduced below in this section). The property $minFTT : S \rightarrow \mathbb{N}^+$ denotes the minimum of the fault-tolerance times of an ASWC for its different safety goals (see also section 2.5.2). The *minFTT* defines how quickly (in ms) a loss of an ASWC has to be recovered by enabling a slave instance of the ASWC to become the new master instance, such that a temporarily loss of the ASWC has no negative effect onto the availability of the realized functional features. The *minFTT* property is used to decide if a hot-standby slave has to be established, or if a cold-standby slave is sufficient, in case $redncy(s) > 0$. The *minFTT* of ASWCs could also be removed from the input model and instead derived from the *minFTT* of the realized features. Property $domain : S \rightarrow \mathbb{N}_0$ defines the functional domain of the ASWC. With this property, it can be controlled that ASWCs are put into different domain specific ASWC-Clusters, like powertrain or chassis domain of a vehicle, although the ASWCs may have identical ASIL and redundancy (fail-operational) requirements. If the domain property is not useful for a system under analysis, it can be set to an identical value for all ASWCs.

We assume that all ASWCs have a single entry-point operation, which becomes executed by a scheduler periodically in fixed execution cycles, and that all ASWCs are executed with the same rate in a common execution cycle (single rate scheduling). See sections 4.9 and 4.10.2 for a discussion about this.

Ports of ASWCs: Each port of an ASWC $s_i \in S$ has an assigned *data-ID* property $d : \mathtt{PP}_i \rightarrow \mathbb{N}^+$ respectively $d : \mathtt{PS}_i \rightarrow \mathbb{N}^+$, defining the output data published by a port, respectively the input data subscribed by a port. Hence, $d(pp_{i,k})$ is the data-ID of the kth publication port of $s_i \in S$, and $d(ps_{i,k})$ is the data-ID of the kth subscription port of $s_i \in S$. A publication port can only be connected to a subscription port by a communication channel, if the assigned data-IDs are identical, meaning $d(pp_{i,k}) = d(ps_{j,l})$, with $i, j, k, l \in \mathbb{N}^+$. We call an instance of a data-ID a *data item*.

For each data-ID, we define how much memory space is required to allocate for data items of the data-ID. We call this amount of memory the *weight* ω and attach it as property to publication ports $\omega : \mathtt{PP}_i \rightarrow \mathbb{N}^+$, denoting the amount of data that is published by a port per execution cycle. Because the data-ID (and thereby the weight) of a publication port and a subscription port being connected by a channel

is identical, it is sufficient to model the weight only for the publication ports. We assume a static amount of data communicated over a channel in each execution cycle, and therefore model a static weight for each publication port. If the amount is not static during runtime, the weight ω contains the maximum data size. We write $\omega(pp_{i,k})$ for the weight of the kth publication port of ASWC s_i. This also denotes the weight of the communication channels that leave a publication port towards connected subscription ports.

In addition, we assign each publication port an identifier $portId : \mathtt{PP}_i \rightarrow \mathbb{N}^+$. We assign the value $portId(pp_{i,k}) = 1000 * i + k$ to each port $pp_{i,k} \in \mathtt{PP}_i$ of ASWC $s_i \in S$, modeled as constraint over the model elements. Within the union set $\bigcup_{s_i \in S} \mathtt{PP}_i$ of all publication ports of all ASWCs, the port-identifiers are unique under the assumption that each ASWC s_i has at most 999 publication ports. To distinguish optional and mandatory subscriptions, each subscription port has a property named $isOptional : \mathtt{PS}_i \rightarrow \{0, 1\}$, which is 1 if it is optional that the subscription port is connected to a publication port, otherwise 0.

Execution Units E: We assume that all execution units have identical synchronous execution cycles (see section 2.5.1). Further we assume that all ASWCs have the same execution rate and hence are executed in every cycle. The property $providedTimeBudget : E \rightarrow \mathbb{N}^+$ defines the budget of time that an execution unit provides in each periodic cycle to execute the cyclic callable operation of deployed instances of ASWCs (in μs). The property $providedFlash : E \rightarrow \mathbb{N}^+$ defines the amount of flash memory that is provided to store binary images of ASWCs (in kilobyte). We do not model other types of memory, like RAM or NVRAM. These can be handled in a similar manner as the time budget and flash. Property $hwPlatform : E \rightarrow \mathbb{N}_0$ defines the hardware platform of an execution unit, usable to distinguish heterogeneous execution units having for instance different microcontrollers. Property $powerSupply : E \rightarrow \{0, 1\}$ defines the power supply to which an execution unit is connected to [0: *Blue* power supply, 1: *Red* power supply]. The reason is that for instance the RACE concept envisages two power supplies to ensure that not the complete set of execution units switches off if a single power-supply fails, enabling fail-operational features even in case of power-supply failures.

4.4.3 Solution Model Properties for Initial Deployment

The solution part of the model are the properties that contain the decision variables of the model, which are determined by the solver during calculation of the solution, following the given constraints. For instance, these are the mapping of ASWCs to ASWC-Clusters, the deployment of ASWCs to execution units, the degradation scenarios resulting from isolations of system elements after assumed failures of these system elements, but also other variables like the sum of the WCETs of all ASWCs that a actively deployed to a certain execution unit.

System Configuration Φ: To the already introduced solution properties *CM*, *CO*, *deploy* and *map* of the system configuration, we add the property $networkTraffic : \mathbb{N}_0$, containing the amount of data which is communicated between ASWCs that are actively deployed to different execution units. Hence, the communication is performed remotely over the network that interconnects the execution units. The amount of remotely communicated data depends on the deployment of ASWCs to execution units (see *deploy*), the selected communication channels between the ASWCs (see *CM* and *CO*), as well as the channel weights ω. We show in section 4.5.1, how the calculation of the network traffic is encoded as a constraint.

Application Software Components S: Each ASWC has a solution property $asil : S \rightarrow \{0..4\}$, defining its *Automotive Safety Integrity Level* (ASIL). The values denote [0: Quality Management (QM), 1: ASIL A, 2: ASIL B, 3: ASIL C, 4: ASIL D]. We derive the ASIL of ASWCs by the maximum of the ASILs of those features, to which an ASWC $s \in S$ contributes to realize, expressed by constraint $asil(s) = max(asil(f) \mid f \in$

$\chi^{-1}(s)$). We do not consider ASIL decompositions [177]. Property $rednc y : S \rightarrow \mathbb{N}_0$ defines the level of redundancy, with which an ASWC has to be (at least passively) deployed to the execution units [$redncy(s) = n$ denotes that $s \in S$ has to be deployed $n+1$ times (passively or actively)]. We derive the redundancy $redncy(s)$ of an ASWC $s \in S$ based on the fail-operational requirements $failOp(f)$ of those functional features f being in the subset $\chi^{-1}(s)$ of functional features to whose realization ASWC s contributes to. However, in addition we limit the redundancy to be at most $|E|-1$, meaning that the maximum number of redundant instances equals the number of execution units. Hence, we calculate the derived redundancy level by $redncy(s) = min\Big(|E|-1, max\big(failOp(f) \mid f \in \chi^{-1}(s)\big)\Big)$, for $s \in S$. See section 4.10.2 for a discussion about why we limit the redundancy by the amount of execution units.

In case $redncy(s) > 0$, it has to be decided if a passive cold-standby *slave* is sufficient or if a hot-standby *slave* is required. The benefit of a hot-standby slave is that its internal state and variables are in sync with the master. A cold-standby slave is started from its initial state and values, leading potentially to a temporary deviation from users expected behavior. A hot-standby slave is active in the schedule, but its output data is ignored by the RTE of the system under analysis. Contrary to this, a cold-standby slave is only passively deployed and not in the schedule. The decision, if a hot-standby slave is required or if a cold-standby is sufficient, is based on the $faultRecoveryTime$ of the system and the *minimum fault-tolerance time* $minFTT(S)$ of the ASWCs for their different safety goals [35], both given as properties of the input problem model. The result is captured in the solution property $hotStandbySlaveReq : S \rightarrow \{0,1\}$.

ASWC-Clusters C: The following properties of ASWC-Clusters $c \in C$ depend on the subset of ASWCs that are mapped to the cluster. The property $asil : C \rightarrow \{0..4\}$ defines the ASIL of a cluster. We ensure by a constraint that all ASWCs within a cluster do have an identical ASIL, and that the ASIL of the cluster is derived by the ASIL of the ASWCs mapped to this cluster. Property $redncy : C \rightarrow \mathbb{N}_0$ defines the redundancy level of a cluster. We ensure by a constraint that all ASWCs within a cluster do have an identical redundancy level, and that the redundancy level of the cluster is derived by the redundancy level of the ASWCs mapped to this cluster. Furthermore, it is ensured by constraints that all ASWCs within a cluster have an identical requirement regarding if a hot-standby slave has to be established, or if a passive cold-standby slave is sufficient (see $hotStandbySlaveReq(s)$).

Communication Channels between ASWCs: More than one matching publication port may exist for a subscription port in a software architecture. In the input problem model, each subscription port is specified by a list of matching publication ports, to which it may be connected. We assume in this thesis that each subscription port shall be connected to at most one publication port. Exactly one communication channel is established that ends at a mandatory subscription port. At most one communication channel is established that ends at an optional subscription port. Constraints ensure that in the solution model, exactly one matching publication port is chosen and a channel is established from the chosen publication port to the concerned subscription port. We now introduce the resulting solution properties of ASWC ports, required to encode the above selection mechanism in the formal model.

Ports of ASWCs: Let $s_i \in S$ be an ASWC that sends data via a set of publication ports $\mathtt{PP}_i$. Let $s_k \in S$ be an ASWC that receives data via a set of subscription ports $\mathtt{PS}_k$, with $i,k \in \mathbb{N}^+$. Each subscription port $ps_{k,l} \in \mathtt{PS}_k$, with $l \in \mathbb{N}^+$, has a solution property called $chosenMatchingPortId : \mathtt{PS}_k \rightarrow \mathbb{N}$, which finally contains the *portId* of the publication port to which $ps_{k,l}$ becomes connected during our synthesis. The design space for the selection of $chosenMatchingPortId(ps_{k,l})$ is the set of the *portId*s of publication ports that have an identical data-ID than $ps_{k,l}$. Formally, this can be expressed as:

$$chosenMatchingPortId(ps_{k,l}) \in \bigcup_{s_i \in S} \; \bigcup_{pp_{i,j} \in \mathtt{PP}_i \,\mid\, d(pp_{i,j}) = d(ps_{k,l})} \Big\{portId(pp_{i,j})\Big\}$$

When modeling this as a formal constraint for the employed Z3 SMT solver, we use an *OR*-clause provided by the Z3 Python API. For each subscription port $ps_{k,l} \in \mathrm{PS}_k$, we setup the following constraint:

$$\bigvee_{s_i \in S} \; \bigvee_{pp_{i,j} \in \mathrm{PP}_i \,|\, d(pp_{i,j}) = d(ps_{k,l})} \mathit{chosenMatchingPortId}(ps_{k,l}) = \mathit{portId}(pp_{i,j})$$

For instance, for two matching publications $pp_{i,j} \in \mathrm{PP}_i$ and $pp_{m,n} \in \mathrm{PP}_m$ of two publisher software components $s_i, s_m \in S$ with identical data-IDs $d(pp_{i,j}) = d(pp_{m,n}) = d(ps_{k,l})$, it holds that

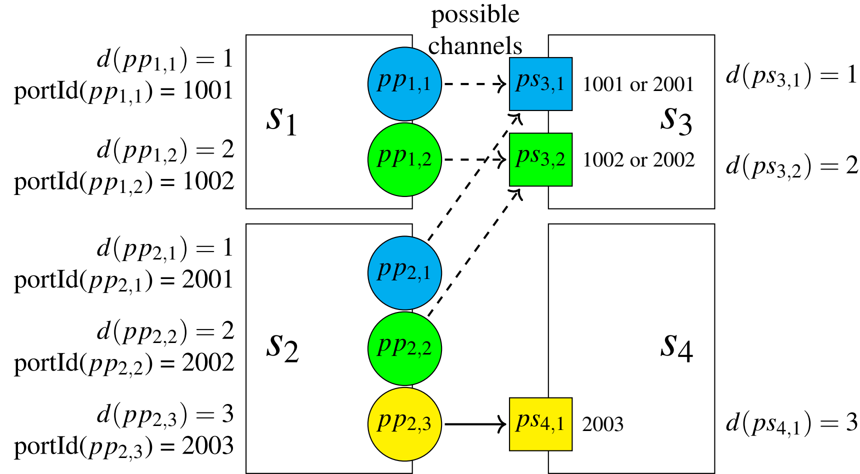
$$\Big(\mathit{chosenMatchingPortId}(ps_{k,l}) = \mathit{portId}(pp_{i,j})\Big) \vee \Big(\mathit{chosenMatchingPortId}(ps_{k,l}) = \mathit{portId}(pp_{m,n})\Big)$$

By using the API of the Z3 SMT solver, this can be encoded as

$$\mathit{Or}\Big(\mathit{chosenMatchingPortId}(ps_{k,l}) = \mathit{portId}(pp_{i,j}), \mathit{chosenMatchingPortId}(ps_{k,l}) = \mathit{portId}(pp_{m,n})\Big)$$

We assume that the determination of matching ports with identical data-IDs is already done before and encoded as part of the input model. Hence, the *OR*-clause constraints defining valid values of *chosenMatchingPortId* can be created during parsing the input model, based on the possible port matches encoded in the input model. However, the resulting communication channels between the matching ports are part of the solution model.

Example for a channel selection: Fig. 4.15 shows an example for a channel selection. The colors of the ports indicate the data-ID of the publication and subscription ports. Ports with the same color have matching data-IDs and can be connected by a communication channel.



Or(chosenMatchingPortId($ps_{3,1}$) = 1001, chosenMatchingPortId($ps_{3,1}$) = 2001)
Or(chosenMatchingPortId($ps_{3,2}$) = 1002, chosenMatchingPortId($ps_{3,2}$) = 2002)
Or(chosenMatchingPortId($ps_{4,1}$) = 2003)

Figure 4.15: Example of four ASWCs with some publications and mandatory subscriptions and possible communication channel candidates

It is shown in the port labels in Fig. 4.15 that $pp_{1,1}$ and $pp_{2,1}$ publish data items of the same data-ID $d(pp_{1,1}) = d(pp_{2,1}) = 1$ (blue port color). As port $ps_{3,1}$ subscribes data items of data-ID $d(ps_{3,1}) = 1$, both of these publication ports could be connected to $ps_{3,1}$. As $portId(pp_{1,1}) = 1001$ and $portId(pp_{2,1}) = 2001$, this choice is encoded by

$$Or\big(chosenMatchingPortId(ps_{3,1}) = 1001, chosenMatchingPortId(ps_{3,1}) = 2001)\big)$$

Also port $ps_{3,2}$ has two matching publisher candidates. Port $ps_{4,1}$ has only one matching publication port $pp_{2,3}$, whose port identifier is $portId(pp_{2,3}) = 2003$. The possible channels of subscription ports $ps_{3,1}$ and $ps_{3,2}$ are dashed, as for each of them only one of both possible channels is chosen. For the given example, four valid channel selections exist, two for $ps_{3,1}$ times two for $ps_{3,2}$. All subscription ports are assumed to be mandatory in this example. Hence, four valid mandatory channel matrices *CM* exist. One of them is selected in combination with the calculation of the deployment.

We discuss the channel selection and alternatives for it in more detail in section 4.10.2. Notice that in the example that was shown in Fig. 4.8 and Fig. 4.12, no such options were contained for choices for the selection of one out of multiple matching publishers.

If more than one matching publication port matches to a subscription, our approach chooses the most suitable publication based on optimization objective definitions. With this, a design space is opened for chosen communication channels and an optimal communication configuration can be configured, choosing local publishers instead of remote publishers to prefer local communication and to minimize network traffic. We introduce the used set of optimization objectives in section 4.8. As the optimality of the channel selection depends on the deployment of ASWCs to execution units, the channel selection and deployment decision are done in unison.

Finally our approach checks if all mandatory subscriptions can be serviced by publications. If not, there is no valid solution to the deployment problem. Even if matching publications are present, it might happen that the deployment constraints in connection with constraint $networkTraffic \leq maxNetworkTraffic$ prohibit a valid solution.

Channel Matrices *CM* and *CO*: Both channel matrices $CM : S \times S \to \mathbb{N}_0$ and $CO : S \times S \to \mathbb{N}_0$ contain in their cells $CM(s_i, s_k)$ resp. $CO(s_i, s_k)$ the aggregated weights $\omega : \mathtt{PP}_i \to \mathbb{N}^+$ of the subset of publication ports of $s_i \in S$ that are connected to mandatory resp. optional subscription ports $\mathtt{PS}_k$ of $s_k \in S$, formally expressed by following equation 4.2.

$$CM(s_i, s_k) = \sum_{ps_{k,l} \in \mathtt{PS}_k \mid isOptional(ps_{k,l})=0} \quad \sum_{pp_{i,j} \in \mathtt{PP}_i \mid \circledast} \omega(pp_{i,j})$$

$$\circledast \equiv chosenMatchingPortId(ps_{k,l}) = portId(pp_{i,j}) \quad (4.2)$$

For $CO(s_i, s_k)$ the equation is similar, but going over the optional subscription ports having property value $isOptional(ps_{k,l}) = 1$. We express equation 4.2 by a formal constraint as shown in constraint $C_{4.1}$ in section 4.5.1.

Execution Units E: For execution units $e \in E$, the amount of used execution time is defined by property $usedTimeBudget : E \to \mathbb{N}_0$ (in μs). For $e \in E$, a constraint ensures that $usedTimeBudget(e)$ becomes the sum of the WCETs $wcet(s)$ of those ASWCs $s \in S$ that are actively deployed onto e. Another constraint ensures that $\forall e \in E : usedTimeBudget(e) \leq providedTimeBudget(e)$. Property $usedFlash : E \to \mathbb{N}_0$ is the amount of flash memory which is occupied by the binaries of the ASWCs that are deployed to execution unit $e \in E$ actively or passively (in kilobyte). A constraint ensures that $usedFlash(e)$ becomes the sum of the required flash memory $flash(s)$ of those ASWCs $s \in S$ that are actively or passively deployed onto e.

4.4.4 Solution Model Properties for Failure Scenarios

In this section, we extend the initial solution properties shown in section 4.4.3 with additional properties with respect to the consideration of degradations and failovers necessary after isolations of failing system elements.

Our objective is to maximize the value of the active ASWCs in a sense that the ASWCs with the highest requirements according to safety ($asil(S)$) and fail-operationality ($redncy(S)$) will be kept active as long as possible. ASWCs with low requirements according to these properties are deactivated first if the system resources become insufficient, for instance after isolations of execution units. To fulfill the mentioned objective, we introduce so called *priority-points* that define the importance of a deployed ASWC instance.

Application Software Components S: In order to analyze degradation scenarios that may appear after failures of ASWCs, property $isolated : S \rightarrow \{0,1\}$ defines if an ASWC $s \in S$ is assumed to be isolated in a considered degradation scenario. It is not a typical solution property, but is set during our analysis depending on the degradation scenario that is analyzed. We introduce in section 4.6 in more detail how we model and analyze degradation scenarios.

To cover degradation scenarios that might be required after isolations of execution units or ASWC instances, each $s \in S$ has the following properties:

- $hotStandbySlaveReq : S \rightarrow \{0,1\}$ indicates if a redundant hot-standby *slave* is required (already introduced in section 4.4.3).
- $hotStandbySlaveActive : S \rightarrow \{0,1\}$ indicates if a required hot-standby *slave* can be kept active. In degradation scenarios, it can be valid that a hot-standby slave is deactivated due to insufficient resources, depending on the $failOp(f)$ properties of the features $f \in \chi^{-1}(s)$ for $f \in F$ and $s \in S$.
- $masterActive : S \rightarrow \{0,1\}$ indicates if one *master* instance can be kept active in a considered degradation scenario. In the initial fault-free scenario, all ASWCs have an active master instance. In degradation scenarios, it is only allowed that no active master instance exists, if the requirements with respect to fail-operational behavior of the realized functional features $f_j \in \chi^{-1}(s)$ are not violated.

In order to decide about the order in which ASWC instances should be deactivated in case of insufficient resources, we assign priority properties to the active ASWC instances. We define a base priority named *prioPoints* for each ASWC. We derive this priority depending on the ASIL and the desired redundancy of an ASWC. The importance of a master and a hot-standby slave is distinguished by introducing two additional instance specific properties *prioPointsMaster* and *prioPointsHotSlave*.

- $prioPoints : S \rightarrow \mathbb{N}_0$ is a property storing the priority of an ASWC. We derive the priority based on $asil(S)$ and $redncy(S)$. The reason for taking the ASIL into account is that as higher the safety integrity level is, the more reliable an ASWC should be, and hence the longer an ASWC should be kept active. The reason for taking the redundancy into account is that it is derived from the fail-operational requirements of those functional features that are realized by the considered ASWC. Hence, we calculate the valuation of $prioPoints(s)$ by $\forall s \in S :$ $prioPoints(s) = asil(s) + redncy(s)$. See section 4.10.2 for a discussion about why we calculate $prioPoints(s)$ like this.
- $prioPointsHotSlave : S \rightarrow \mathbb{N}^+$ is the priority of the *hot-standby slave* instance of an ASWC. A constraint ensures that the value becomes $prioPointsHotSlave(s) = prioPoints(s) + 1$, but only if a hot-standby slave is required ($hotStandbySlaveReq(s) = 1$), otherwise we set $prioPointsHotSlave(s) = 0$.

- $prioPointsMaster : S \rightarrow \mathbb{N}^{+}$ is the priority of the *master* instance of an ASWC. The master instances are more important than hot-standby slave instances. Hence, we define a constraint ensuring that $\forall s \in S$: $prioPointsMaster(s) = prioPoints(s) + 2$.

Execution Units E: The property $isolated : E \rightarrow \{0,1\}$ defines if an execution unit $e \in E$ is assumed to be faulty and isolated in a considered degradation scenario. At system runtime, the isolations will be performed in case a failure of that execution unit has been detected. We introduce in section 4.6 in more detail how we model and analyze degradation scenarios.

Functional Features F: Each functional feature has a solution property $available : F \rightarrow \{0,1\}$, defining if a feature $f \in F$ is still available in a considered degradation scenario.

System Level Properties in System Configuration Φ: On system configuration level, the property $prioSumAllASWCs : \mathbb{N}_0$ is defined as the sum of the priorities of the actively deployed ASWCs in the initial deployment without any isolation. Property $prioSumActiveASWCs : \mathbb{N}_0$ is the sum of the priority-points of all ASWC instances that are actively deployed in the current system degradation scenario, either as master (using $prioPointsMaster(s)$ in the sum) or as hot-standby slave ($prioPointsHotSlave(s)$). In degradation scenarios, some ASWC instances may be deactivated, forcing *prioSumActiveASWCs* to become smaller than *prioSumAllASWCs*. The more ASWC instances become deactivated, the smaller *prioSumActiveASWCs* becomes, and potentially also the more functional features become unavailable, leading to a higher system degradation. During the analysis, one objective is to maximize *prioSumActiveASWCs* in all degradation scenarios. Hence, features with no fail-operational requirement and a low ASIL will become unavailable first, as the realizing ASWCs will be deactivated first.

Auxiliary System Level Properties: Furthermore, we add the following auxiliary properties to the model. We use these properties in the formal constraints to avoid repeated recalculations of the values in different constraints.

- $amountOfIsolatedExecUnits : \mathbb{N}_0$ — amount of isolated execution units, being the amount of those $e \in E$ having property $isolated(e) = 1$.
- $amountOfIsolatedSWCs : \mathbb{N}_0$ — amount of ASWCs which are in state *isolated* (see section 4.4.5), being the amount of those $s \in S$ having property $isolated(s) = 1$.
- $activeBlueExecUnits : \mathbb{N}_0$ — amount of active execution units, attached to the *blue* power supply. This is the amount of those $e \in E$ having properties $isolated(e) = 0 \wedge powerSupply(e) = 0$.
- $activeRedExecUnits : \mathbb{N}_0$ — amount of active execution units, attached to the *red* power supply. This is the amount of those $e \in E$ having properties $isolated(e) = 0 \wedge powerSupply(e) = 1$.

4.4.5 States of ASWC Instances

Fig. 4.16 shows the different states that ASWC instances might have during runtime, as well as possible transitions between these states. The cardinalities in the states denote the number of instances of an ASWC that might have this state simultaneously. For instance, at any time at most one master and at most one hot-standby slave exists, but several cold-standby slaves may exist.

Notice that in the deployment solution property $deploy(s,e)$ we do not distinguish the passive states. This means that for all the passive states ('cold-standby slave', 'deactivated/inactive' and 'isolated') it holds

that $deploy(s,e) = P$. This is sufficient to analyze the system degradation in failure scenarios, and reduces the complexity. However, the both active states ('master' and 'hot-standby slave') are distinguished in $deploy(s,e)$, as this is necessary to analyze the systems degradation scenarios. In the figures, like in the examples in sections 4.6.8, 4.6.9 and 4.7.5, we always color the masters in green and the hot-standby slaves in yellow, to make it easier to distinguish the different component states.

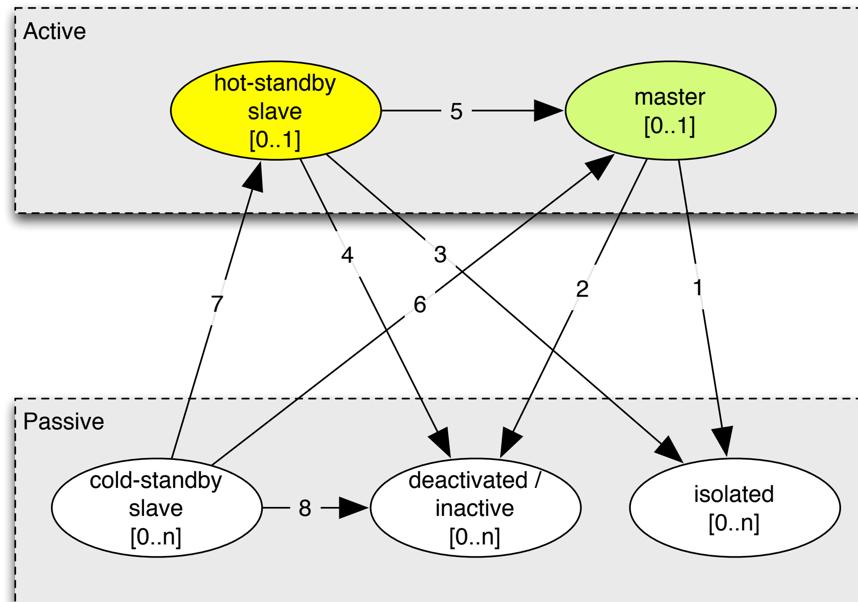


Figure 4.16: Software component states

The state transitions in Fig. 4.16 are as follows:

1. A master becomes *isolated*, if a failure of the master ASWC instance has been detected, to avoid a propagation of this failure.

2. A master becomes explicitly *deactivated*, if either not enough resources are left to execute it on the execution unit to which the master was deployed to, or if mandatory input data items are missing due to the deactivation or isolation of some other ASWC instance.

3. A hot-standby slave becomes *isolated*, if a failure of this slave ASWC instance has been detected. Although the outputs of a hot-standby slave are ignored by the RTE and not forwarded to any receiver, these outputs can be monitored by the RTE to detect failures.

4. A hot-standby slave becomes explicitly *deactivated*, if either not enough resources are left to execute it on the execution unit to which it was deployed to, or if mandatory input data items are missing due to the deactivation or isolation of some other ASWC instance.

5. A hot-standby slave can become a master, if the former master became isolated or deactivated or was lost due to the isolation of the execution unit which accommodated it.

6. A cold-standby slave can become a master, if the former master became isolated or deactivated or was lost due to the isolation of the execution unit which accommodated it. This is only valid, if $hotStandbySlaveReq(s) = 0$, otherwise always the hot-standby slave has to become the new master.

7. A cold-standby slave can become a hot-standby slave, if the former hot-standby slave became a master or isolated or deactivated.

8. A cold-standby slave becomes explicitly *deactivated*, if mandatory input data items are missing such that the cold-standby slave can never be activated anymore due to these missing inputs.

During the transitions shown in Fig. 4.16, we assume that failures (and isolations) are permanent and hence, e. g., missing data items do not reappear.

A 'cold-standby slave' and a 'deactivated/inactive' ASWC will never be explicitly isolated, as no failure can occur within them because they are not executed and only exist in memory as binary. However, if the execution unit, onto which an ASWC instance is passively or actively deployed, has a hardware failure and has to be isolated, then also these ASWC instances are gone. This holds for every of the shown states. Hence, we do not model the 'gone' state explicitly, instead we say that the ASWCs that were used to be deployed onto an isolated execution unit have no state, as the whole execution unit is isolated.

4.5 Synthesis of Valid Redundant Deployments

Depending on the chosen safety and redundancy concept, there might exist a huge set of requirements with respect to the validity of deployments of ASWCs to execution units, and with respect to the validity of failover and degradation scenarios. We formalize such requirements by formal constraints, ensuring the validity of the calculated solutions during our analysis.

Below we introduce how we model formal constraints that formally express informal requirements for valid redundant deployments. All these constraints must be fulfilled by a valid deployment. We assume the requirements given from the safety and redundancy concept introduced in section 2.5.2. The redundancy is built into the deployments to be prepared for possible failure scenarios, which become introduced next in section 4.6, to ensure fail-operational requirements of functional features.

Syntax used in the formal constraints: The Constraints are shown in a notation close to the API of the Z3 SMT solver, but somehow shortened for space reasons. To define the constraints, we use two conditional functions. The first function `If(I,T,E)` is an *if-then-else* definition. Parameter `I` describes an *if-clause*. If `I` is true, then parameter `T` is used in the constraint, else parameter `E`. The second function that we use is the implication `Implies(I,T)`, which is like an `If` without the *else* part `E`. The implication is `true` for $(\neg \texttt{I} \vee \texttt{T})$. In addition, we use a `sum` operation, calculating the sum over the listed elements. For instance, $sum_{e \in E}(usedTimeBudget(e))$ is the sum over the used time budgets of all execution units. We use `sum` instead of sign $\sum$ to avoid confusions with the set of scenarios Σ (see Section 4.6) in the constraints. Often, we build the sum over conditional statements, using the `If(I,T,E)` operation within the sum. All functions are provided by the used SMT-Solver.

4.5.1 Formal Constraints for Valid Redundant Deployments

In this section, we provide some examples showing how informal requirements for the redundant deployments can be ensured by formal constraints over the formal system model, which we introduced in section 4.2. We write R_x to reference *Requirement x*, and C_y to reference *Constraint y*. Comments are annotated with '|' in the listings that show the formal constraints.

Notice that for completeness, in the following constraints we already use the Failure Scenario parameter $\sigma \in \Sigma$, which we are going to introduce subsequently in Section 4.6 (Def. 15).

Requirement R_1 contains information about redundant deployments. It contains sub-requirements that are ensured separately by constraints $C_{1.1}$, $C_{1.2}$ and $C_{1.3}$.

Requirement 1 ASWCs with a required redundancy level of n have to be deployed $n+1$ times. From these redundant instances, at most 2 instances are active, 1 as master, and if required 1 as hot-standby slave. All other redundant instances are passive initially and may be activated in degradation scenarios.

Constraint $C_{1.1}$ ensures the correct number of allocations of ASWCs to execution units. The sum is calculated over a conditional statement. For every passive or active instance of an ASWC $s \in S$ on an execution unit $e \in E$, expressed by $deploy(s,e,\sigma) \neq 0$, a *one* is added to the sum, else a *zero*. ASWCs with $redncy(s) = n$ have to be allocated $n+1$ times (passively or actively).

Constraint 1.1 (R_1):

```
1 ∀s ∈ S : ∀σ ∈ Σ :
2 sum_{e∈E}( If(deploy(s,e,σ) ≠ 0,1,0)) = redncy(s)+1          | degree of redundancy
```

Constraint $C_{1.2}$ controls the amount of master instances of each ASWC. If a master instance is present, it is ensured that it exists exactly once. To deactivate a master, property $masterActive(s,\sigma)$ has to become 0. Analyzing the value of $masterActive(s,\sigma)$ enables to give feedback to the user that ASWC s cannot be executed in the currently considered degradation scenario σ. The hot-standby slaves are handled similarly in constraint $C_{1.3}$.

Constraint 1.2 (R_1):

```
1 ∀s ∈ S : ∀σ ∈ Σ :
2 Implies( masterActive(s,σ) = 1,                               | if a master is active
3         sum_{e∈E}( If(deploy(s,e,σ) = M,1,0)) = 1             | then exactly 1 master
4 )
```

Constraint 1.3 (R_1):

```
1 ∀s ∈ S : ∀σ ∈ Σ :
2 Implies( hotStandbySlaveActive(s,σ) = 1,                      | if a hot slave is active
3         sum_{e∈E}( If(deploy(s,e,σ) = HS,1,0)) = 1            | then exactly 1 hot slave
4 )
```

Furthermore, we use the value of $masterActive(s,\sigma)$ together with the realization relationship $\chi^{-1}(s)$ to determine which functional features can be kept available, see constraint $C_{7.1}$ in section 4.6.6 below. This enables to give feedback to the users about the availability of features in the degradation scenarios.

Requirement 2 The master and the hot-standby slave of an ASWC must not be deployed onto two execution units that are attached to the same power supply. This is required to avoid that master and hot-standby slave disappear simultaneously in case one of the two power supplies of the assumed system breaks down. However, in degradation scenarios, not in the initial deployment, there exist exceptions. If already one power supply failed, or if every single execution unit attached to one power supply failed, then the master and the hot-standby slave of an ASWC can be deployed onto two execution units attached to the same power supply.

Constraint 2.1 (R_2):

```
1  ∀s ∈ S : ∀σ ∈ Σ :
2  Implies(                                                    | if all the following is true:
3    And( hotStandbySlaveReq(s) = 1,                           |  hot-slave is required
4         hotStandbySlaveActive(s,σ) = 1,                      |  hot-slave is active
5         masterActive(s,σ) = 1,                               |  master is active
6         activeBlueExecUnits(σ) > 0,                          |  at least 1 blue exec-unit is alive
7         activeRedExecUnits(σ) > 0                            |  at least 1 red exec-unit is alive
8    ),                                                        | then:
9    And(sum_{e∈E}( If(And(powerSupply(e) = 0, deploy(s,e,σ) ∈ {M,HS}),1,0)) = 1,
10       sum_{e∈E}( If(And(powerSupply(e) = 1, deploy(s,e,σ) ∈ {M,HS}),1,0)) = 1
11   )
12 )
```

Constraint $C_{2.1}$ ensures the one aspect of valid initial deployments, described in R_2. With $deploy(s,e,\sigma) \in \{M,HS\}$ we express an active deployment, either as master (M) or as hot-standby slave (HS). Furthermore, $C_{2.1}$ contains the described exception. The exception is handled by the two auxiliary properties 'activeBlue-ExecUnits' and 'activeRedExecUnits', which were introduced in section 4.4.4. These properties count the amount of active execution units, attached to the *blue* respectively the *red* power supply. Hence, constraint $C_{2.1}$ must only hold if at least one *blue* and one *red* execution unit is active. The situation, if either all red or all blue execution units are isolated, is handled by an additional constraint, not shown here.

Requirement 3 The amount of remote communication data, communicated over the network, must not exceed a given threshold per execution cycle. Hence, deployments should be determined that prefer local communication.

Constraint $C_{3.1}$ calculates the amount of network traffic for each scenario, arising from remote communication channels between active ASWCs deployed to different execution units. Notice that the published output data of hot-standby slaves is assumed to be not sent to the subscribers, but only the published output data of masters (see line 5). But the subscribed input data must be provided for both hot-standby slaves and masters (line 6-7). If the communication between a master of a publisher and the hot-standby slave or the master of a subscriber is remote, then the weight of the corresponding mandatory and optional communication channels has to be added to the network traffic (line 10). Constraint $C_{3.2}$ ensures that the network traffic never exceeds a given threshold.

Constraint 3.1 (R_3):

```
1  ∀σ ∈ Σ :
2  networkTraffic(σ) = sum_{s_i,s_k∈S} sum_{e_x,e_y∈E} (
3      If(                                                     | if the following is true:
4          And( e_x ≠ e_y,                                     |  two different execution units
5               deploy(s_i,e_x,σ) = M,                         |  s_i is master on e_x
6               Or( deploy(s_k,e_y,σ) = M,                     |  s_k is master on e_y
7                   deploy(s_k,e_y,σ) = HS                     |  or s_k is hot-slave on e_y
8               )
9          ),
10         CM(s_i,s_k) + CO(s_i,s_k),                          | then: channels from s_i to s_k use network
11         0                                                   | else: channels from s_i to s_k are local
12     )
13 )
```

Constraint 3.2 (R_3):

```
1 ∀σ ∈ Σ:
2 networkTraffic(σ) ≤ maxNetworkTraffic
```

Another constraint ensures that optional channels are only removed if the publisher ASWC has to be deactivated due to a failure or insufficient resources, but not removed just to decrease network traffic.

Notice that in the example shown in section 4.6.9, the solution was calculated without the above objective to minimize the remote network communication. When we activate the minimize objective for the network traffic, we can analyze how the degradation behavior changes.

Requirement R_4 defines the values of the cells in the mandatory channel matrix. The requirement for the optional channel matrix is omitted here, as it is very similar.

Requirement 4 The mandatory channel matrix $CM : S \times S \to \mathbb{N}_0$ shall contain in its cells $CM(s_i, s_k)$ the aggregated weights $\omega : \mathtt{PP}_i \to \mathbb{N}^+$ of the subset of publication ports $\mathtt{PP}_i$ of $s_i \in S$ that are connected to mandatory subscription ports of $\mathtt{PS}_k$ of $s_k \in S$.

Constraint $C_{4.1}$ ensures that the solution model adheres to requirement R_4. The constraint is equivalent to equation 4.2, introduced in section 4.4.3.

Constraint 4.1 (R_4):

```
1  ∀s_i, s_k ∈ S:
2  CM(s_i, s_k) = sum_{pp_{i,j}∈PP_i} sum_{ps_{k,l}∈PS_k} (
3      If(                                                  | if the following is true:
4          And(
5              chosenMatchingPortId(ps_{k,l}) = portId(pp_{i,j}),   | pp_{i,j} connected to ps_{k,l}
6              isOptional(ps_{k,l}) = 0                     | ps_{k,l} is mandatory
7          ),
8          ω(pp_{i,j}),                                     | then: add weight ω(pp_{i,j})
9          0                                                | else: add nothing
10     )
11 )
```

4.5.2 Examples

We show later in sections 4.6.8, 4.6.9 and 4.7.5 three examples. For each example we show synthesized valid redundant deployments, each both for an initial failure free situation, as well as several follow-up deployments for assumed failure scenarios.

4.6 Analysis of Failure Effects

In this section, we introduce our basic concept of failure effect analysis. We assume permanent failures of execution units and software components. We assume that these failures are detected by the RTE of the system under analysis, and that the RTE provides a mechanism to isolate the failed elements from the residual system (see also section 4.9.2). We consider different *failure scenarios* and analyze the resulting level of degradation, the fulfillment of all fail-operational requirements of functional features, and needed failovers between redundantly deployed software components.

4.6.1 Scenarios

We extend the formal model by so called *scenarios*, representing different situations that may appear in the system runtime. We use the scenarios to express different situations of failing system elements. The failing system elements become isolated by the RTE of the system. If the failure scenario leads to a degradation of the available set of functional features of the system, we also call it *degradation scenario*. If a failover has to be performed in a scenario to keep available a fail-operational functional feature, we also call it *failover scenario*. Combined, we also call it *degradation/failover scenario*.

Definition 15 – Scenarios: We define Σ as a finite set of identifiers of scenarios, and $\sigma \in \Sigma$ as a single identifier of a scenario.

This means, the finite set $\Sigma = \{\sigma_0, \sigma_1, ..., \sigma_m\}$, with $m \in \mathbb{N}$, contains the scenarios that we consider during our analysis. Element $\sigma_0 \in \Sigma$ represents the initial scenario without any assumed failure and hence, without any isolation. The $\sigma_i \in \Sigma$ with $i \in \mathbb{N}^+$ represent the scenarios in which some execution units and/or software components are assumed to fail and become isolated, forcing consequently degradations and/or failovers to may be necessary. We define $\Sigma^+ = \Sigma \setminus \{\sigma_0\}$ to be the set of potential degradation/failover scenarios. However, notice that due to the analysis result, it may be the case that for some $\sigma \in \Sigma^+$ no degradation or failover is necessary, depending on the assumed failure in a scenario and the synthesized deployment of software components to execution units.

4.6.2 The Scenario Graph

Based on the finite set Σ of scenarios, we construct possible transitions between the scenarios.

Definition 16 – Transitions between Scenarios: We define $T : \Sigma \times \Sigma^+ \rightarrow \{0, 1\}$ as a transition relation between two scenarios.

A scenario transition $T(\sigma_i, \sigma_k)$ represents a transition from a scenario $\sigma_i \in \Sigma$ towards a successor scenario $\sigma_k \in \Sigma^+$, with $i \in \mathbb{N}_0$ and $k \in \mathbb{N}^+$. If $T(\sigma_i, \sigma_k) = 1$, there exists a transition from σ_i to σ_k, otherwise not.

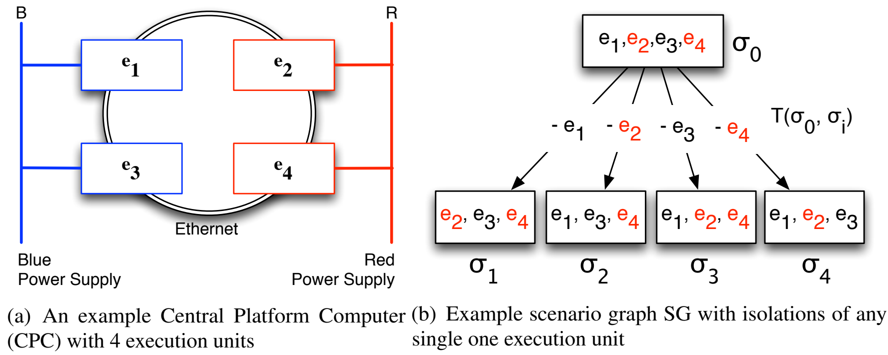
As we do not considerer healing of already isolated system elements, the transitions build a directed acyclic graph. We call this the *scenario graph*, as introduced below.

Definition 17 – Scenario-Graph (SG): Let the finite set Σ represent the nodes (alias vertexes) of a graph. Let the scenario transitions T represent the edges between the nodes of a graph. We introduce a *Scenario-Graph* (SG) as a directed acyclic graph $SG = (\Sigma, T)$.

For the scenario nodes Σ, we introduce a property $label : \Sigma \rightarrow \{\mathcal{P}(E \cup S)\}$. The label values are constructed as follows. Let $EF \subset E$ be the set of isolated failed execution units in a specific hardware failure scenario $\sigma \in \Sigma$. Let $SF \subset S$ be the set of isolated failed software components in a specific software failure scenario $\sigma \in \Sigma$. This means that $EF = \{e \in E \mid isolated(e) = 1\}$ and $SF = \{s \in S \mid isolated(s) = 1\}$. The label property represents for each graph node the union set of non-isolated execution units $EA = E \setminus EF$ and non-isolated software components $SA = S \setminus SF$.

The edges T describe transitions between scenario nodes. A transition in SG happens due to an isolation of a failed execution unit, or due to an isolation of a failed software component, meaning that an $e \in E$ moves from EA to EF, respectively a $s \in S$ moves from SA to SF. Also the scenario transitions have a label property $label : T \rightarrow \{E \cup S\}$, representing the execution unit respectively the software component, that is assumed to fail at this transition.

Example: Isolation Scenarios of Execution Units: Let us consider a set of scenarios of failing execution units. We ignore failures of software component in this example. Fig. 4.17(a) shows an example hardware architecture with four execution units [3], which are by example attached to two different power-supplies (red and blue). When considering only one subsequently isolation, we obtain a set of five scenario nodes $\Sigma = \{\sigma_0, \sigma_1, \sigma_2, \sigma_3, \sigma_4\}$. The scenario graph SG looks like shown in Fig. 4.17(b). The nodes are labeled with the set of alive execution units $EA \subseteq E$. The edges are labeled with that execution unit $e \in E$, which has recently been failed and isolated. The red color in the text in the node labels denote those execution units which are attached to the red power-supply.



(a) An example Central Platform Computer (CPC) with 4 execution units

(b) Example scenario graph SG with isolations of any single one execution unit

Figure 4.17: Example Scenario-Graph (SG)

We consider the scenario in which a power-supply (R,B) breaks down, leading to the simultaneous disappearance of the entire subset of execution units attached to this power-supply, as a common-cause failure of all of these execution units. This means, we set the $isolated(e_i)$ flag to true for all execution units that are attached to the power-supply which is assumed to fail. Based on this, the failure effect analysis is performed. Hence, we consider a breakdown of a power-supply as multiple isolations of all attached execution units. Otherwise, our definition of levels of fail-operationality cannot be applied, which says that if $failOp(f) > 0$, then feature $f \in F$ must be kept available with full-fledged functionality during the first $failOp(f)$ failures of any one execution unit or ASWC (see section 4.4.2). If we would consider a power-supply breakdown as a single failure during our analysis, this definition would not hold, as potentially multiple redundant spare components disappear simultaneously in case of a power-supply breakdown.

During the failure effect analysis, we calculate the deployment matrix *deploy* for different scenarios of isolations of execution units or ASWCs. In case of a required degradation due to insufficient execution resources or mandatory input data, master instances of ASWCs are deactivated, meaning that entries of matrix *deploy* with value $deploy(s,e) = M$ disappear. By analyzing the values of the matrix *deploy* and the realization relationships χ^{-1} between ASWCs and functional features in the nodes of the scenario graph, the consequential level of system degradation for each scenario can be determined by analyzing which features can be kept available.

During our analysis, we define the members of Σ usually as all scenarios up to a certain amount of isolations, but Σ could also be defined as part of the input model.

[3]This hardware architecture conforms to a Central Platform Computer (CPC) according to the system architecture concept developed in the RACE project (see section 2.5)

Example: Degradation after an isolation of an execution unit: For a second example, lets assume a set of only two execution units $E = \{e_1, e_2\}$. When considering only failures of execution units and ignoring failures of ASWCs in this example, we obtain a set of three scenarios $\Sigma = \{\sigma_0, \sigma_1, \sigma_2\}$ and two scenario transitions $T(\sigma_0, \sigma_1)$ and $T(\sigma_0, \sigma_2)$. In scenario σ_1, execution unit e_1 is assumed to fail and to become isolated. In σ_2, this is assumed for e_2.

Fig. 4.18 shows how an initial deployment in scenario σ_0 may alter in case of a scenario transition $T(\sigma_0, \sigma_2)$. During the transition, a failover is applied to keep a fail-operational feature available, and a degradation is applied due to a necessary deactivation of another feature because of insufficient resources. In the successor scenario σ_2, execution unit e_2 becomes isolated, due to an assumed detected hardware failure of that unit. Two functional features $F = \{f_1, f_2\}$ exist, realized by three ASWCs $S = \{s_1, s_2, s_3\}$ (as also the case in the example in Fig. 4.10 and Fig. 4.11). Feature f_2 has a fail-operational requirement $failOp(f_2) = 1$, feature f_1 has no fail-operational requirement.

In σ_2, the passive cold-standby slave of ASWC s_3 on e_1 has to be activated, because the former master on e_2 is lost. Assuming that s_1, s_2 and s_3 cannot run simultaneously on e_1 due to resource constraints, s_1 and s_2 become passivated to free resources to execute s_3. This is allowed as s_1 and s_2 realize features that have no requirement to be available after a failure ($\chi^{-1}(s_1) = \{f_1\}$, $\chi^{-1}(s_2) = \{f_1\}$ and $failOp(f_1) = 0$). As s_1 and s_2 become passivated, feature f_1 cannot be provided anymore and becomes unavailable.

We mark the lost system elements with a red cross. The system element, which was initially isolated due to a detected failure, is marked with a black flash symbol, like e_2 in Fig. 4.18.

Notice that all requirements concerning fail-operationality are met in this example. It is ensured by formal constraints in our model, that no ASWC instance is active anymore on an isolated execution unit.

4.6.3 Extensions of Model Properties to Cover the Scenarios

In order to analyze the different scenarios $\sigma \in \Sigma$, we add the parameter Σ to all solution properties that vary in the different scenarios, see listing 4.1.

```
Functional Features:
    available : F × Σ → {0,1}

ASWCs:
    hotStandbySlaveActive : S × Σ → {0,1}
    masterActive : S × Σ → {0,1}
    isolated : S × Σ → {0,1}

Execution Units:
    usedTimeBudget : E × Σ → ℕ0
    isolated : E × Σ → {0,1}

System Configuration:
    deploy : S × E × Σ → {0,P,M,HS}
    prioSumActiveASWCs : Σ → ℕ0
    networkTraffic : Σ → ℕ0
    amountOfIsolatedExecUnits : Σ → ℕ0
    amountOfIsolatedSWCs : Σ → ℕ0
    activeBlueExecUnits : Σ → ℕ0
    activeRedExecUnits : Σ → ℕ0
```

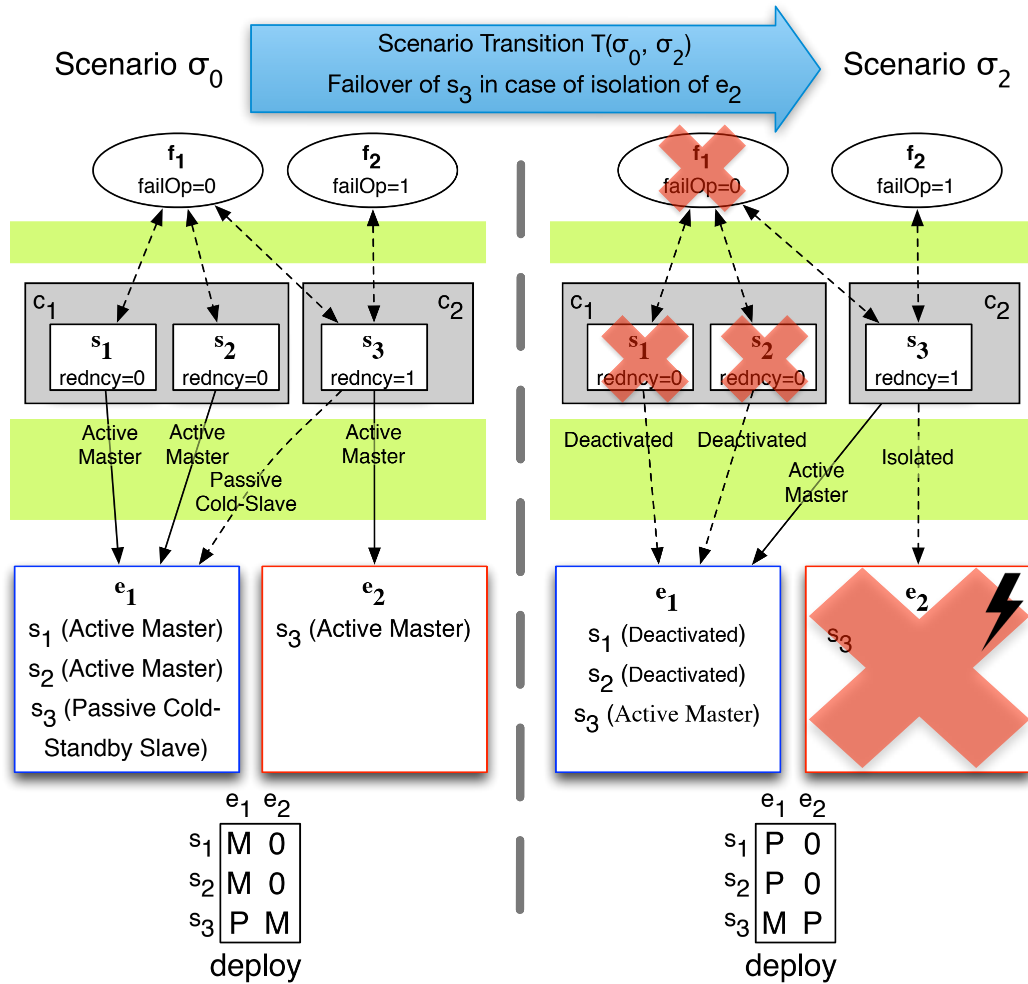


Figure 4.18: Example of a system degradation after an execution unit isolation

Listing 4.1: Solution properties that get extended by scenario identifier Σ

This means, the respective properties introduced in sections 4.4.3 and 4.4.4 become extended with the scenario parameter, as their value may become different in different scenarios.

4.6.4 Procedure to Analyze the Scenario Graph

Each node in the scenario graph SG represents a degradation scenario, or a failover scenario, or a combination of both. We talk about degradation/failover scenarios. When analyzing these scenarios, we have two options.

1. We could solve each scenario separately by calculating one solution for each node of the graph by the SMT solver. When doing this, we have to save some solution properties of the initial deployment

and fix them for the solutions of the degraded follow-up deployments. This is necessary to avoid undesired effects at the transitions between the graph nodes, like undesired migrations of ASWCs between execution units. The benefit is that we can obtain valid solutions for all nodes where this is possible, and maybe no result for nodes where the solver identifies the input problem as unsatisfiable. However, the drawback is that solving and analyzing each graph node separately offers a quite bad total performance. The reason is that we need to add an algorithm that traverses all graph nodes, setup the constraints for the actual node in the SMT input model, solve the model, parse the solution, store those solution properties that have to be taken over to the next graph node, traversing to the next graph node, setup a new SMT input model, and so on and so forth. Even though the Z3 SMT solver provides push() and pop() operations for models, therewith a common base model can be kept and not always the entire model has to be constructed from scratch, this is quite inefficient.

2. We can calculate the whole scenario graph at once. To do this, we have to avoid undesired changes at the transitions by defining appropriate additional constraints. Those solution properties which vary in different scenarios, like the deployment of ASWCs and the availability of functional features, have to be extended by a scenario identifier to distinguish the different solutions for the different scenarios. We showed in previous section 4.6.3 which properties are concerned by this. The benefit is that the solutions can be calculated more efficient. The drawback is that as soon one node in the graph becomes unsatisfiable, the whole graph problem becomes unsatisfiable, as long as we do not introduce so called *soft constraints* that relax some constraints such still a solution for the whole graph problem can be found, but indicating that for some nodes not all constraints could be hold strongly. We show in section 4.6.7, how soft constraints can be modeled.

4.6.5 Formal Constraints for Valid Failovers and Degradations

In section 4.5.1, we introduced the basic formal constraints that ensure the calculation of valid redundant deployments. In this section, we introduce additional constraints for valid degradation and failover scenarios.

The formal model is solved for the initial situation σ_0 and also for the degradation scenarios σ_j with $j \in \mathbb{N}^+$. For each scenario transition $\sigma_i \to \sigma_k$, with $i \in \mathbb{N}_0$ and $k \in \mathbb{N}^+$, some solution properties of the previous scenario σ_i are used as fixed properties for the follow-up scenario σ_k, in order to avoid undesired changes in the solution, like migrations of ASWCs between execution units. This is ensured by constraints.

Before we start, we introduce how we calculate the auxiliary properties for the degradation scenarios, which were introduced in section 4.4.4. The values of these properties are be calculated as follows.

```
1  ∀σ ∈ Σ:
2    amountOfIsolatedExecUnits(σ) = sum_{e∈E}(isolated(e,σ))
3
4    amountOfIsolatedSWCs(σ) = sum_{s∈S}(isolated(s,σ))
5
6    activeBlueExecUnits(σ) =
7        sum_{e∈E}(If(And(powerSupply(e) = 0, isolated(e,σ) = 0), 1, 0))
8
9    activeRedExecUnits(σ) =
10       sum_{e∈E}(If(And(powerSupply(e) = 1, isolated(e,σ) = 0), 1, 0))
```

We now discuss some requirements for valid degradations, as well as how we ensure them by formal constraints over the introduced system model.

Requirement 5 If the execution unit, to which the master instance of an ASWC is deployed to, becomes isolated due to a failure, and if a hot-standby slave was active on another execution unit before, then the hot-standby slave becomes the new master. No other inactive cold-standby slave should become the new master in this case.

Constraint $C_{5.1}$ ensures requirement R_5. The constraint describes the transition $T(\sigma_i, \sigma_k)$ from a scenario $\sigma_i \in \Sigma$ to a successor scenario $\sigma_k \in \Sigma^+$.

Constraint 5.1 (R_5):

```
1  ∀s ∈ S:
2  ∀e_x, e_y ∈ E:
3  ∀σ_i, σ_k ∈ Σ with T(σ_i, σ_k) = 1:
4  Implies(                                   | if all the following is true:
5      And( deploy(s, e_x, σ_i) = M,          |  s is master on e_x in scenario σ_i
6           deploy(s, e_y, σ_i) = HS,         |  s is hot slave on e_y in scenario σ_i
7           isolated(e_x, σ_k) = 1,           |  e_x becomes isolated in scenario σ_k
8           isolated(e_y, σ_k) = 0,           |  e_y is still operating in scenario σ_k
9           masterActive(s, σ_k) = 1          |  master of s is still active in scenario σ_k
10     ),                                     | then:
11     deploy(s, e_y, σ_k) = M                |  s on e_y becomes new master in σ_k
12 )
```

Similar requirements and formal constraints exist for the activation of a new hot-standby slave, and for the switch of a cold-standby slave to become the new master, in case no hot-standby slave is required. We do not show them here, as $C_{5.1}$ shows the principle how this can be encoded in formal constraints.

Other constraint exist, e. g., to ensure that the master instance of an ASWC does not migrate to another execution unit in a follow-up deployment during a scenario transition, if the execution unit to which the master was initially deployed is still alive. Also the definition of the degradation scenarios, meaning which execution unit and which software component is set to $isolated(e, \sigma) = 1$ respectively $isolated(s, \sigma) = 1$ in which scenario $\sigma \in \Sigma$, is done by a constraint. Furthermore, additional constraints might be desired to be added to optimize the degradation scenarios. For instance, if an ASWC s_i is active, but all features $f_j \in \chi^{-1}(s_i)$ are already unavailable because some other ASWCs $s_k \in \chi(f_j)$ became deactivated, and if no other required ASWC depends mandatorily on data published by s_i, then also s_i can be deactivated in order to make efficient use of resources. Another optimization to make efficient usage of resources in degradation scenarios is to deactivate the hot-standby slave of a fail-operational feature, if the feature is allowed to become unavailable after the next failure of a system element. The hot-standby slave is no more mandatorily in this case.

In addition to this, we formalize certain constraints to calculate helper properties, required to perform our analysis. One of these is the property *prioSumActiveASWCs*, as described below.

Requirement 6 As a helper requirement for our analysis approach, we require for each considered failure scenario the calculation of a sum of the priority points of those application software components which are actively deployed in that scenario, either as master or as hot-standby slave.

Constraint $C_{6.1}$ ensures the calculation of a property named *prioSumActiveASWCs* for each failure scenario $\sigma \in \Sigma$, which represents requirement R_6.

Constraint 6.1 (R_6):

```
1  ∀σ ∈ Σ:
2  prioSumActiveASWCs(σ) = sum_{s∈S} sum_{e∈E}(
```

```
3         If (  deploy(s,e,σ)  = M,                  | if s is master on e in scenario σ
4              prioPointsMaster(s),                  | then add prioPointsMaster(s)
5              If (  deploy(s,e,σ)  = HS,            | else if s is hot-slave on e in scenario σ
6                   prioPointsHotSlave(s),           | then add prioPointsHotSlave(s)
7                   0                                | else add 0
8              )
9         )
10   )
```

4.6.6 Formal Constraints to Analyze Feature Availability

Below we describe how we trace between ASWCs and functional features in the constraints, allowing to analyze the effects of isolations or deactivations of ASWCs onto the availability of the realized functional features in each degradation scenario.

Requirement 7 If at least one of the ASWCs $s \in \chi(f)$, which realize a feature $f \in F$, has no active master instance anymore in the current degradation scenario $\sigma \in \Sigma$, then the feature f has to be marked as non-available in σ.

Constraint 7.1 (R_7):

```
1 ∀f ∈ F : ∀s ∈ χ(f) : ∀σ ∈ Σ:
2    Implies(masterActive(s,σ) = 0, available(f,σ) = 0)
```

Analyzing the value of *available*(f,σ) enables to give feedback to the user about the availability of functional feature f in scenario σ, and thereby to determine the level of required degradation of the set of available functional features in scenario σ. Another constraint ensures that features are only allowed to become non-available, if their fail-operational requirement does not become violated due to this.

In addition, we model an optimization objective, defining that the sum of available features has to be maximized in each scenario. Hence, during calculating the solutions, the solver only sets those *available*(f,σ) to 0, for which it is necessary due to constraint $C_{7.1}$. See section 4.8 for the definitions of the optimization objectives.

Requirement 8 A feature $f \in F$ is only allowed to become non-available in a degradation scenario, if the number of already appeared failures of system elements (execution units and/or ASWCs) is higher than the fail-operational level *failOp*(f) of the feature. This means, only after at least $x+1$ isolations of execution units and/or ASWCs, a functional feature f having *failOp*$(f) = x$ is allowed to become inactive. Here, isolations of ASWCs due to detected failures at their interface-behavior are meant, not explicit deactivations of ASWCs due to insufficient resources. However, one exception are degraded features as introduced in section 4.7. A degraded feature f' is always non-available as long as the related feature $f = \text{degf}^{-1}(f')$ is available.

Constraint $C_{8.1}$ handles the availability of full-fledged features, and constraint $C_{8.2}$ handles the availability of degraded features. Both take care to adhere to the fail-operational requirements.

Constraint 8.1 (R_8):

```
1 ∀f ∈ F : ∀s ∈ χ(f) : ∀σ ∈ Σ:
2 Implies(                                              | if all the following is true:
3      And  (  degf⁻¹(f) = −1,                          | f is not a degraded feature
4              failOp(f) ≥ (amountOfIsolatedExecUnits(σ)
```

```
5                          + amountOfIsolatedSWCs(σ) )
6      ),                                                    | then:
7      available(f,σ) = 1                                    | f must be available in σ
8  )
```

As the solver has to assign a value to each property, we use the value -1 to represent the $\perp$ of the codomain $\text{degf}^{-1} : F \to F \cup \{\perp\}$, as no functional feature has the identifier -1.

Constraint 8.2 (R_8):

```
1  ∀f, f′ ∈ F : ∀σ ∈ Σ:
2  Implies(                                                  | if all the following is true:
3      And  (degf(f) = f′,                                   | f′ is the degraded version of f
4            degf⁻¹(f′) = f,                                 | f is the full-fledged version of f′
5            available(f,σ) = 0,                             | f is no more available in σ
6            failOp(f′) ≥ (amountOfIsolatedExecUnits(σ)
7                         + amountOfIsolatedSWCs(σ) )
8      ),                                                    | then:
9      available(f′,σ) = 1                                   | f′ must be available in σ
10 )
```

The above shown informal requirements and related formal constraints show only a subset of the constraints necessary to perform our failure effect analysis approach.

The motivation is to illustrate how formal constraints can be expressed based on our formal model. Depending on the requirements for redundant deployments, failover mechanisms, valid degradation scenarios, etc, the constraints can be adapted to the needs of the requirements of the system under analysis.

4.6.7 Relaxation of Constraints to Localize Problems

In case the input problem has an incorrect design that does not enable to fulfill all fail-operational requirements in all degradation scenarios, it is beneficial to obtain an indication about which parts of the input model, like which functional features or which ASWC instances on which execution units, are the reason for the problem.

If the whole set of constraints are *hard* constraints, meaning that they must be fulfilled in order to obtain any valid solution from the solver, then the solver would return an *unsat* for the problem, denoting that the problem was unsatisfiable. This gives no clue about the reason for why no solution was found.

Hence, we classify some constraints as soft constraints to allow the solver to ignore these constraints, if this is necessary to find any valid solution. However, if this happens, we need an information about which constraints cannot be satisfied, in order to identify the set of related system elements and give feedback about parts of the architecture that might be required to become improved.

Traditional SMT solvers without optimization support are able to return a so called *unsatisfiable core (unsat core)*, if not all constraints can be satisfied [83]. The unsatisfiable core contains the constraints that hinder the solver to find a valid solution. To be helpful, the unsatisfiable cores are aimed to be as small as possible.

However, the recent version of the Z3 SMT solver with support of optimization objectives [52] does not support *unsatisfiable cores* anymore, as there would not be any way to satisfy objectives when the hard constraints are unsatisfiable. Hence, we model a subset of the constraints as soft constraints, using implications from boolean variables onto constraints.

We define a boolean constraint tracking property $\mathit{softConstraintTracker} : \mathbb{N} \times \Sigma \rightarrow \{0,1\}$, where the first parameter $i \in \mathbb{N}$ is an unique natural number identifier of a soft constraint (without $\forall$ quantifiers), and the second parameter denotes the considered degradation scenario.

$$\text{Implies}\Big(\underbrace{\text{softConstraintTracker}(i,\sigma) = 1}_{\textit{antecedent}},\ \underbrace{\text{<Constraint>}}_{\textit{consequent}}\Big)$$

The following requirement and soft constraint gives a simple example.

Requirement 9 If an ASWC is required to be deployed redundantly and if a hot-standby slave is required for this ASWC, then the hot-standby slave *should be* active as long as the amount of isolated execution units is smaller than the required level of redundancy.

If the amount of isolated execution units equals the redundancy level of the ASWC ($\mathit{redncy}(s) = \mathit{amountOfIsolatedExecUnits}(\sigma)$), then it may be the case that only one single instance of the ASWC is left (as master or deactivated). However, due to insufficient resources, it may happen that the hot-standby slave becomes deactivated earlier, in order to keep active master instances of other ASWCs. This should be recognizable by the analysis approach.

Given an ASWC $s \in S$, a degradation scenario $\sigma \in \Sigma$, and a natural number $i \in \mathbb{N}$ that is unique in the scope of scenario σ, we define the soft constraint $C_{9.1}$ for requirement R_9. As the constraint naturally contains an implication, we embed the tracking property into the antecedent of the implication using an *and* operator. We could have also modeled a new implication around the present implication.

Constraint 9.1 (R_9):

```
1 Implies(
2     And (softConstraintTracker(i,σ) = 1,
3          redncy(s) > (amountOfIsolatedExecUnits(σ)
4     ),
5     hotStandbySlaveActive(s,σ) = hotStandbySlaveReq(s)
6 )
```

Hence, if the original constraint without the tracker cannot be satisfied in a scenario, then the solver can set the tracker to become 0, with the effect that the constraint is satisfied. Due to this, the solver is able to return a valid solution, in which the tracker is set to 0. The set of boolean tracker properties being equal to 0 enables to identify the soft constraints that could not be satisfied in the solution, and via this to identify the architecture elements (e. g., the software components) that are involved in the constraints. The set of unsatisfied soft constraints is also called the *correction set* [51] [49].

In order to avoid that trackers of soft constraints are set to 0 by the solver without the explicit justification that the problem would be unsatisfiable otherwise, we define an objective to minimize the amount of soft constraint trackers having value 0. In other words, we maximize the amount of trackers having value 1.

```
1 ∀σ ∈ Σ:
2 maximize( sum_{i∈ℕ} (If(softConstraintTracker(i,σ) = 1,1,0)) )
```

However, there exist also other optimization objectives as part of our analysis approach. When defining multiple optimization objectives for a problem, a trade-off between these objectives has to be accepted. We show in section 4.8 how we handle the multiple objectives. The above mentioned maximization for the softConstraintTracker is represented by objective O_6 in section 4.8.

4.6.8 Example A – Basic Example

In this section we show the applicability of our approach on a first simplified example, applied to an automotive context. The example was also more briefly discussed in [38].

Input Problem Properties for the Example: Table 4.2 shows a set of functional features, as well as a set of ASWCs realizing these features, inclusive certain input model properties.

Table 4.2: Example set of functional features and the realizing ASWCS with some of the predefined properties

Feature f_i	**asil**(f_i)	**failOp**(f_i)	**ASWCs** s_i **of** $\chi(f_i)$	**asil**(s_i)	**redncy**(s_i)	**wcet**(s_i) **in ms**
f_1 : Infotainment	QM	0	s_1 : Infotainment	QM	0	2.0
f_2 : Energy-Management	A	0	s_2 : RemainingRangeCalc	A	0	0.7
			s_3 : EnergyEfficiencyAssist	A	0	0.3
f_3 : ADAS-A	C	0	s_4 : AdasSwc1	C	0	1.7
			s_5 : AdasSwc2	D	1	1.0
f_4 : ADAS-B	D	1	s_5 : AdasSwc2	D	1	1.0
f_5 : Manual-Driving	D	3	s_6 : ManualAcceleration	D	3	1.0
			s_7 : ManuelBraking	D	3	1.0
			s_8 : ManualSteering	D	3	0.5

The realization relationship $\chi(f_i)$ between functional features and ASWCs is visualized in Fig. 4.19. Notice that s_5 contributes to realize two features f_3 and f_4. Due to this, the ASIL of s_5 is derived as the maximum ASIL of these two features, which is ASIL D. Furthermore, as $failOp(f_4) = 1$, the redundancy is $rednc y(s_5) = 1$.

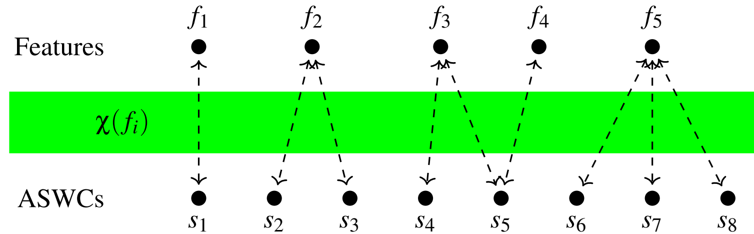


Figure 4.19: Realization relationship $\chi(f_i)$ between functional features and ASWCs for the example of Table 4.2

The features f_3 and f_4 are placeholders for some Advanced Driver Assistance Systems (ADAS), like an ACC or automatic parking. Feature f_4 is required to be kept available after one failure ($failOp(f_4) = 1$), but f_3 is not required to be kept available after one failure ($failOp(f_3) = 0$). As ASWC s_5 contributes to realize both f_3 and f_4, it has $rednc y(s_5) = 1$ to provide enough redundancy required to ensure the fail-operational behavior of f_4. As ASWC s_4 only realizes f_3, it is sufficient that $rednc y(s_4) = 0$.

In this example, five ASWC-Clusters $\{c_1,...,c_5\}$ are established. The cluster mapping matrix *map* is shown below. Empty cells represent a zero. Notice that s_5 is only in one cluster, although it contributes to two features.

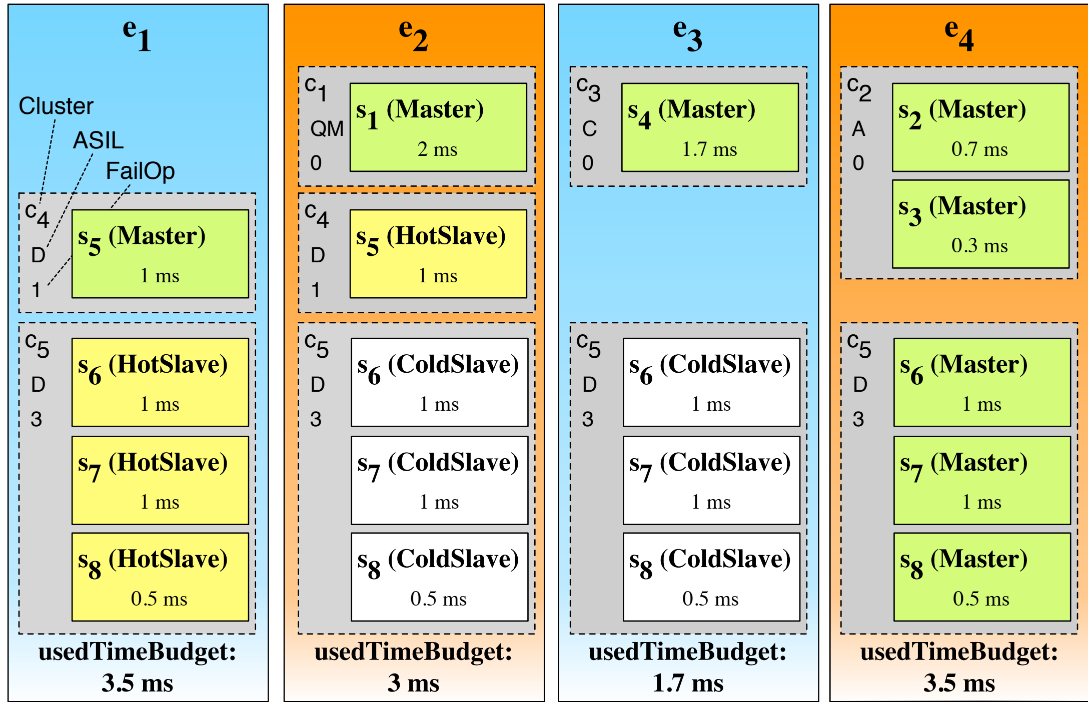
Table 4.3: Cluster mapping matrix $map(s_i, c_j)$

	c_1	c_2	c_3	c_4	c_5
s_1	1				
s_2		1			
s_3		1			
s_4			1		
s_5				1	
s_6					1
s_7					1
s_8					1

We consider a hardware architecture with four execution units, like shown in Fig. 4.17(a). As provided execution time per cycle of the execution units $e_i \in E$, we assume $totalTimeBudget(e_i) = 4ms$. We assume in this example that $minFTT(s_i) \leq faultRecoveryTime$ for all ASWCs $s_i \in S$. Hence, if redundant instances of an ASWC are required, there has to exist a hot-standby slave ($hotStandbySlaveReq(s_i) = 1$).

Calculated Initial Deployment (Scenario σ_0): In the initial deployment (scenario σ_0), all master instances and all required hot-standby slave instances can be deployed, see Fig. 4.20. The colors (red/blue) of the execution units denote their attached power-supply. The colors (blue/yellow/white) of the ASWC instances denote their state (cf. Fig. 4.16 in section 4.4.5). The labels of the ASWCs s_i contain their WCET in milliseconds *ms*. The labels of the ASWC-Clusters c_i contain the ASIL (middle value, e. g., QM or D) and the redundancy level (bottom value, e. g., 0 or 1) of the ASWCs of that cluster.

Notice that $\forall e_i \in E : usedTimeBudget(e_i, \sigma_0) \leq totalTimeBudget(e_i) = 4ms$.



prioSumAllASWCs: 77
prioSumActiveASWCs(σ_0): 77

Deactivated required hot-standby Slaves: --
Deactivated Masters: --
Deactivated Features: --

Figure 4.20: Initial deployment for the example of Tab. 4.2 in scenario σ_0

The sum of priority points in the initial solution is 77. This value is composed by single values as shown in Table 4.4. For instance, $prioPointsHotSlave(s_6) = 4+3+1 = 8$ because it is an ASIL D component (+4), has a redundancy level of 3, and is a hot-standby slave (+1).

Table 4.4: Calculation of $prioSumActiveASWCs(\sigma_0)$

s_i	$prioPointsMaster(s_i)$	$prioPointsHotSlave(s_i)$
s_1	2	—
s_2	3	—
s_3	3	—
s_4	5	—
s_5	7	6
s_6	9	8
s_7	9	8
s_8	9	8
	$\sum 47$	$\sum 30$

$\sum 77$

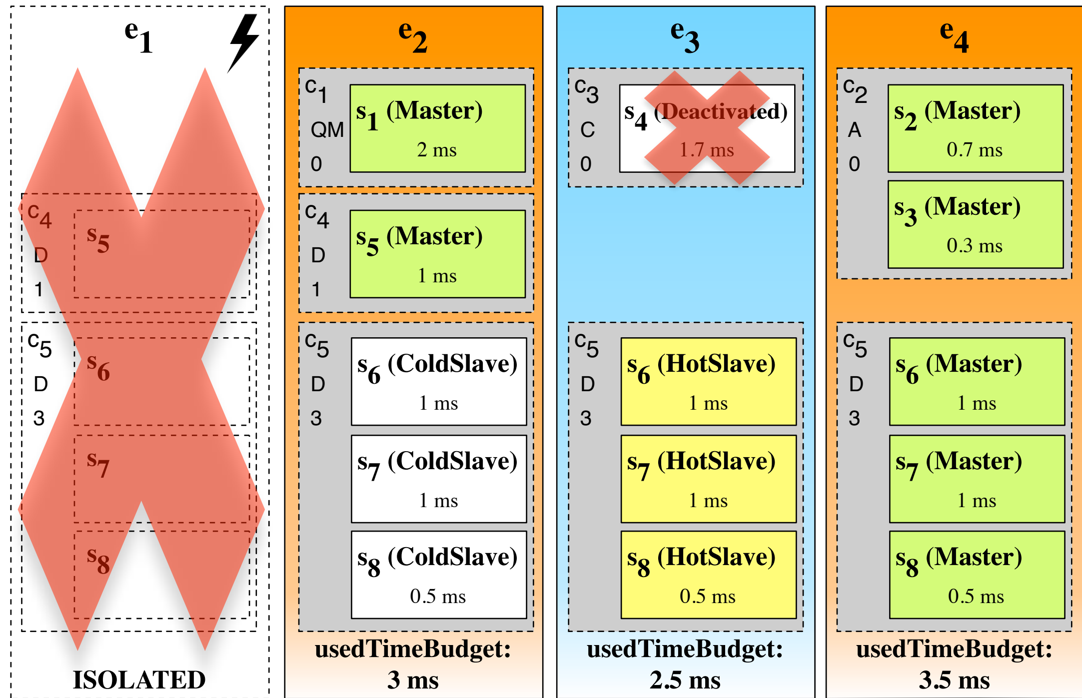
Calculated Degraded Deployment (Scenario σ_1): If for instance the first execution unit e_1 fails and has to be isolated (scenario σ_1), the deployment has to be changed, see Fig. 4.21. After the isolation of e_1, the master of s_5 gets lost and its hot-standby slave on e_2 becomes the new master. As $rednc y(s_5) = 1$, no new slave is created as the redundancy levels describe the required levels of redundancy in the initial failure-free case. A disappearance of s_5 would have had effect on the functional features f_3 and f_4, as $\chi^{-1}(s_5) = \{f_3, f_4\}$. Because $failOp(f_3) = 0$ and $failOp(f_4) = 1$, this disappearance would have been invalid, as the requirements of feature f_4 would be violated. But as a slave existed and became a new master, it is okay. No new slave is required, because it is okay that f_4 is lost after the second isolation of an execution unit. Due to this, it is not required that s_5 is still present after the next isolation. Furthermore, cluster c_5 on e_1 is lost and with it, all three contained hot-standby slaves of ASWCs. It is required that new hot-standby slaves of these ASWCs are created. As $rednc y(s_{6,7,8}) = 3$, an inactive instance of $s_{6,7,8}$ must be activated to serve as new hot-standby slave to prepare for the next isolation. The new slaves of $s_{6,7,8}$ can only be activated on e_3 and not on e_2, because master and hot-standby slave must not depend on the same power-supply. This is to avoid that both the master and the hot-standby slave are lost simultaneously, if the power-supply breaks down. However, to be able to execute $s_{6,7,8}$ on execution unit e_3, ASWC s_4 has to be deactivated as the sum of the WCETs of s_4 and $s_{6,7,8}$ is $4.2ms$, which would exceed the time-budget of $4ms$ of e_3. The deactivation of ASWC s_4 forces the deactivation of feature f_3, as $s_4 \in \chi(f_3)$. This means, if execution unit e_1 has to be isolated, feature f_3 cannot be provided anymore and becomes unavailable, using the given initial deployment that was shown in Fig. 4.20. Notice that a valid initial deployment is calculated automatically by our approach, but it can also be changed manually in order to analyze degradation scenarios depending on different initial deployments.

The loss of the master of s_4 and the hot-standby slave of s_5 forces a loss of 11 priority points, because $prioPointsMaster(s_4) = 5$ and $prioPointsHotSlave(s_5) = 6$. Hence, when execution unit e_1 is isolated, only 66 priority points can be provided by the system (see Fig. 4.21 and Table 4.5).

Table 4.5: Calculation of $prioSumActiveASWCs(\sigma_1)$ after isolation of e_1

s_i	$prioPointsMaster(s_i)$	$prioPointsHotSlave(s_i)$
s_1	2	—
s_2	3	—
s_3	3	—
s_4	—	—
s_5	7	—
s_6	9	8
s_7	9	8
s_8	9	8
	$\sum 42$	$\sum 24$
	$\sum 66$	

If more execution units are isolated in arbitrary order in the shown example, cluster c_5 always remains to have a master instance for all contained ASWCs $\{s_6, s_7, s_8\}$, even if only one execution unit is left. This is important as $failOp(f_5) = 3$ and $\chi(f_5) = \{s_6, s_7, s_8\}$.



prioSumAllASWCs: 77
prioSumActiveASWCs(σ_1): 66

Deactivated required hot-standby Slaves: s_5
Deactivated Masters: s_4
Deactivated Features: f_3

Figure 4.21: Deployment after isolation of execution unit e_1 in scenario σ_1

The deployment matrices $deploy(S,E,\sigma_0)$ and $deploy(S,E,\sigma_1)$ for the both shown scenarios are shown in Fig. 4.22. The changes of values are marked in bold.

	e_1	e_2	e_3	e_4
s_1	0	M	0	0
s_2	0	0	0	M
s_3	0	0	0	M
s_4	0	0	M	0
s_5	M	HS	0	0
s_6	HS	P	P	M
s_7	HS	P	P	M
s_8	HS	P	P	M

deploy(S,E,σ_0)

	e_1	e_2	e_3	e_4
s_1	0	M	0	0
s_2	0	0	0	M
s_3	0	0	0	M
s_4	0	0	**P**	0
s_5	**P**	**M**	0	0
s_6	**P**	P	**HS**	M
s_7	**P**	P	**HS**	M
s_8	**P**	P	**HS**	M

deploy(S,E,σ_1)

Figure 4.22: Deployment matrices for scenarios σ_0 and σ_1 for the example shown in Tab. 4.2

Table 4.6 shows the content of the property *available* : $F \times \Sigma \to \{0,1\}$ for the two shown scenarios σ_0 and σ_1. The functional feature f_3 is not available anymore in σ_1 (see also Fig. 4.21). This is okay as $failOp(f_3) = 0$. Important is that $available(f_4, \sigma_1) = 1$ and $available(f_5, \sigma_1) = 1$, due to the fail-operational requirements of these two features: $failOp(f_4) = 1$ and $failOp(f_5) = 3$.

Table 4.6: Availability of functional features in the two shown scenarios for the example shown in Tab. 4.2

f_i	$available(f_i, \sigma_0)$	$available(f_i, \sigma_1)$
f_1	1	1
f_2	1	1
f_3	**1**	**0**
f_4	1	1
f_5	1	1

4.6.9 Example B – Communication Channels

We now present another example. The system under analysis has 8 ASWCs and the ASWC properties as shown in Fig. 4.23. In this example, we add communication channels and the resulting channel matrices *CM* and *CO* into consideration. The example was also discussed in a briefer manner in [40].

For simplicity, in this example we assign each ASWC its own ASWC-Cluster. Hence, $\alpha(c_i) = \{s_i\}$. Also for simplicity, we assume here that each ASWC realizes a separate functional feature. This means, there are 8 functional features and it holds that $\chi(f_i) = \{s_i\}$.

Fig. 4.23 shows the set of ASWCs and their properties, inclusive published and subscribed data items. The functional features are not shown.

ASWC	asil(s_i)	redncy(s_i)	$\alpha(s_i)$	Hot/Cold standby Slave	wcet(s_i) in ms	Published data items	data item weights $\omega(pp_{i,k})$	Mandatorily Subscribed data items	Optionally Subscribed data items
s_1	D	1	c_1	hot	1.5	d_1, d_2	1, 2	-	-
s_2	D	1	c_2	cold	2.5	-	-	d_1	-
s_3	B	0	c_3	-	2	d_3	3	-	-
s_4	A	0	c_4	-	2	d_4	4	-	-
s_5	QM	0	c_5	-	2	-	-	d_2, d_8	d_4
s_6	QM	0	c_6	-	1	d_5, d_6	5, 6	d_3	-
s_7	QM	0	c_7	-	1	d_7	7	d_4	d_3
s_8	QM	0	c_8	-	1	d_8	8	d_7	d_6

Figure 4.23: Example set of ASWCs with published and subscribed data items d_i

The example contains two ASWCs s_1 and s_2 which have requirements to be deployed with redundancy and hence, to behave fail-operational. The redundancy of ASWCs is derived from the assumed fail-operational requirement of the realized functional features, meaning that $failOp(f_1) = 1$ and $failOp(f_2) = 1$.

For all other f_j with $j \geq 3, failOp(f_j) = 0$. Let a hot-standby slave be required for s_1 (see `hot` in column 5) and a cold-standby slave be sufficient for s_2 (see `cold` in column 5).

Fig. 4.24 shows the component architecture for the example inclusive the published data items (circles) an subscribed data items (squares). The optional subscription ports are dashed and marked with label 'opt'. Notice that published data item d_5 is not subscribed by any component. It is indicated in the labels of s_1 and s_2, that these components will be deployed with redundancy, s_1 with a master and a hot-standby slave $(M+HS)$, and s_2 with a master and a cold-standby (alias passive) slave $(M+P)$.

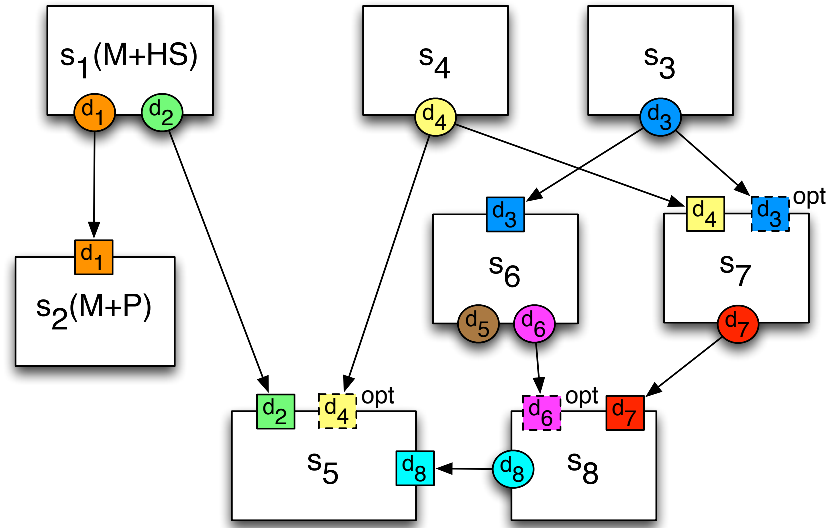


Figure 4.24: Component architecture with communication channels

Based on the given publication ports $\mathtt{PP}_i$ of a $s_i \in S$ and subscription ports $\mathtt{PS}_k$ of a $s_k \in S$, as well as the weights $\omega(pp_{i,j})$ of published data items of $pp_{i,j} \in \mathtt{PP}_i$, the matrix $CM(s_i, s_k)$ of mandatory data-flow communication channels is calculated as shown in Table 4.7.

Table 4.7: Matrix of mandatory channels $CM(s_i, s_k)$ for the example

$s_i \diagdown s_k$	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8
s_1	0	**1**	0	0	**2**	0	0	0
s_2	0	0	0	0	0	0	0	0
s_3	0	0	0	0	0	**3**	**0**	0
s_4	0	0	0	0	**0**	0	**4**	0
s_5	0	0	0	0	0	0	0	0
s_6	0	0	0	0	0	0	0	**0**
s_7	0	0	0	0	0	0	0	**7**
s_8	0	0	0	0	**8**	0	0	0

Those entries of $CM(s_i, s_k)$ equal to 0 are bold, which are unequal to 0 in $CO(s_i, s_k)$.

Table 4.8 shows the matrix $CO(s_i, s_k)$ of optional channels.

Table 4.8: Matrix of optional channels $CO(s_i, s_k)$ for the example

$s_i \diagdown s_k$	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8
s_1	0	0	0	0	0	0	0	0
s_2	0	0	0	0	0	0	0	0
s_3	0	0	0	0	0	0	**3**	0
s_4	0	0	0	0	**4**	0	0	0
s_5	0	0	0	0	0	0	0	0
s_6	0	0	0	0	0	0	0	**6**
s_7	0	0	0	0	0	0	0	0
s_8	0	0	0	0	0	0	0	0

Calculated Initial Deployment (Scenario σ_0): A valid initial deployment of the ASWCs onto four execution units $\{e_1, ..., e_4\}$ is shown in Fig. 4.25. In this example, each unit has $totalTimeBudget(e_i) = 4ms$. In this figure, the ASWC-Clusters are not shown, only the ASWCs. Notice that s_1 is deployed twice, once as a master on e_1 and once as a hot-standby slave on e_2. The hot-standby slave is executed, hence its $wcet(s_1)$ is part of the $usedTimeBudget(e_2, \sigma_0)$. However, the sent data items of the hot-standby slave are ignored by the RTE and not sent over the network. Due to this, the channels that leave the hot-standby slave of s_1 are dashed. ASWC s_2 is also deployed twice, but has a cold-standby slave on e_4, which is not executed.

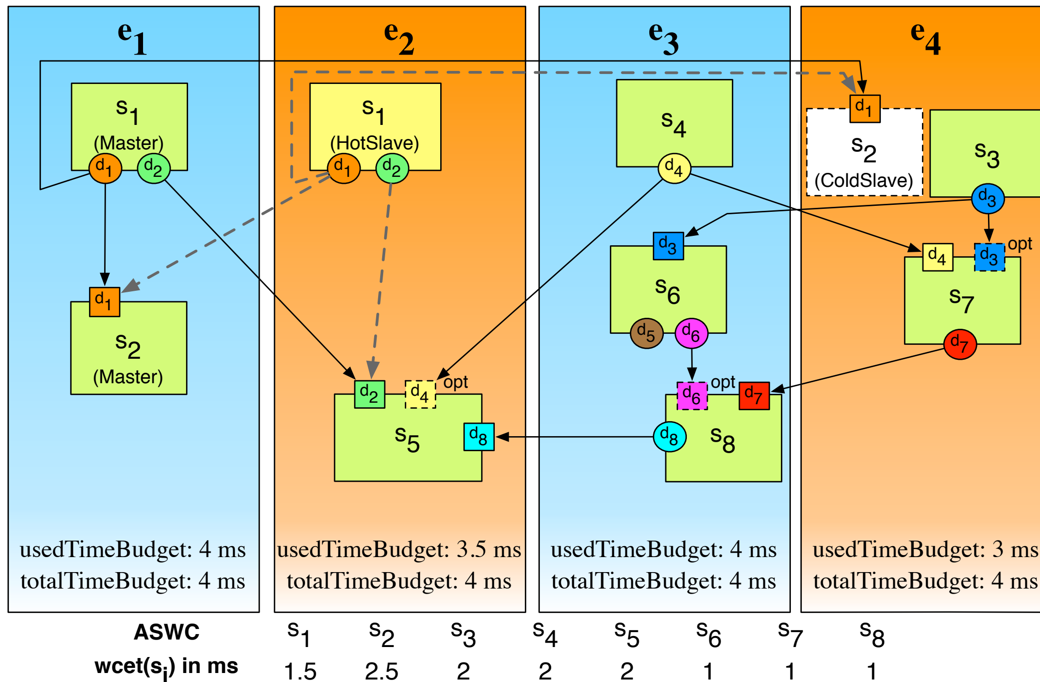


Figure 4.25: Initial deployment for the example of Fig. 4.23 in scenario σ_0

The calculation of $prioSumActiveASWCs(\sigma_0)$ is shown in Table 4.9, for the example shown in Fig. 4.23 in the initial scenario σ_0.

Table 4.9: Calculation of $prioSumActiveASWCs(\sigma_0)$

s_i	$prioPointsMaster(s_i)$	$prioPointsHotSlave(s_i)$
s_1	7	6
s_2	7	—
s_3	4	—
s_4	3	—
s_5	2	—
s_6	2	—
s_7	2	—
s_8	2	—
	$\sum 29$	$\sum 6$
	$\sum 35$	

Calculated Degraded Deployment (Scenario σ_1): Fig. 4.26 shows how the deployment from Fig. 4.25 is reconfigured in case that execution unit e_1 has been isolated after the detection of a hardware failure (scenario σ_1). As the master instance of s_1 on e_1 disappears, the hot-standby slave instance of s_1 on e_2 has to become the new master. Notice that now the communication channels leaving s_1 on e_2 are no more ignored. In addition, the master instance of s_2 on e_1 disappears. As $rednc y(s_2) = 1$, s_2 can still be provided after this isolation, as a new master of s_2 can be activated. In this example, s_2 has only a passive cold-standby slave on e_4. This has to be activated. However, in order to be able to activate s_2 on e_4, one of the other components which are active on e_4 has to be deactivated, as otherwise $usedTimeBudget(e_4, \sigma_1)$ would be bigger than $totalTimeBudget(e_4)$. This forces the deactivation of s_3. Now, another fact has to be considered concerning the established communication channels. As the data item d_3 can no more be published by s_3, all components that mandatorily subscribe this data item cannot fulfill their service anymore. In the example, this is the case for s_6. Hence, the data items d_5 and d_6 can as well no more be published and s_6 can be deactivated to save resources. In the example, this has no effect, as d_5 is not subscribed at all and d_6 is only once subscribed optionally. As a result, Fig. 4.26 shows the subset of components and their communication channels that can still be provided after the isolation of e_1. Notice that all requirements concerning fail-operationality are met in this example, as s_1 and s_2 are still active and hence, features f_1 and f_2 can be kept available.

As before, we mark lost system elements with a red cross. The system element which was isolated due to an assumed detected failure is marked with a black flash symbol (see e_1).

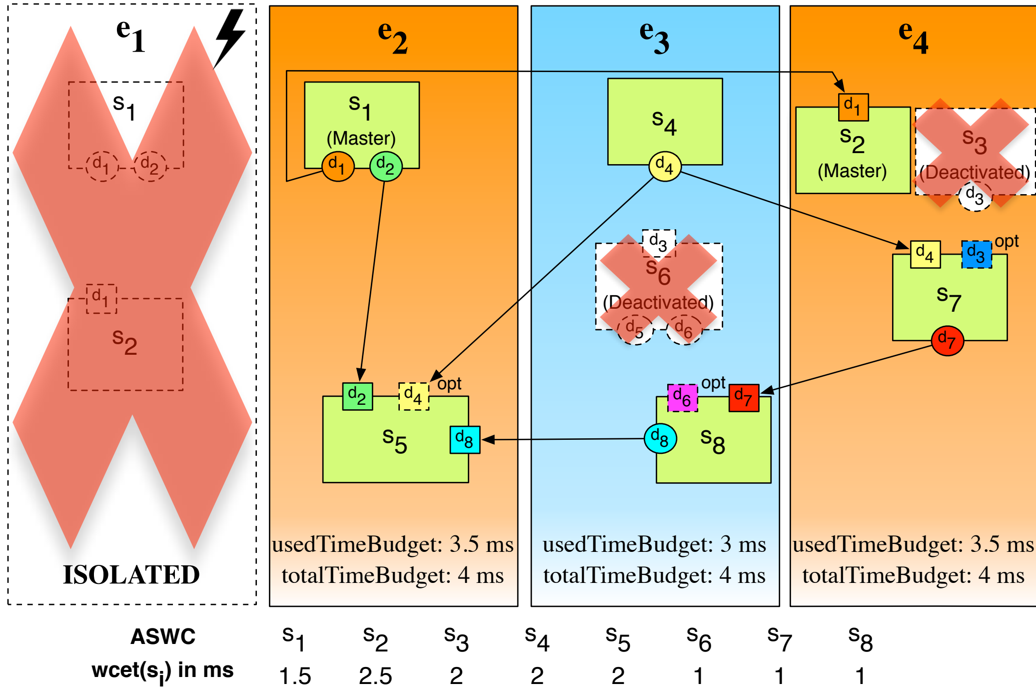


Figure 4.26: Deployment after isolation of execution unit e_1 in scenario σ_1

Table 4.10 shows the calculation of $prioSumActiveASWCs(\sigma_1)$, in the scenario σ_1 considering the isolation of e_1.

Table 4.10: Calculation of $prioSumActiveASWCs(\sigma_1)$ after isolation of e_1

s_i	$prioPointsMaster(s_i)$	$prioPointsHotSlave(s_i)$
s_1	7	—
s_2	7	—
s_3	—	—
s_4	3	—
s_5	2	—
s_6	—	—
s_7	2	—
s_8	2	—
	$\sum 23$	$\sum 0$
	$\sum 23$	

As in the degraded scenario σ_1 it holds that $masterPresent(s_3) = 0$ and $masterPresent(s_6) = 0$, the user gets direct feedback about which ASWCs can no more be actively deployed in this degradation scenario. This supports to analyze the degradation of the availability of functional features in the different scenarios. In this example, this means that features f_3 and f_6 become unavailable in scenario σ_1.

Furthermore, it can be seen that s_1 and s_2 are still active but it cannot be guaranteed anymore that s_1 or s_2 survive the next isolation of an execution unit. This is okay, as s_1 and s_2 were required to survive only

one isolation ($rednc y(s_1) = rednc y(s_2) = 1$). If for instance $rednc y(s_1)$ would have been bigger than 1, additional passive instances would have been deployed for s_1, from which one would have been selected as new hot-standby slave, allowing s_1 to survive more than 1 isolations of execution units.

The deployment matrices $deploy(S,E,\sigma_0)$ and $deploy(S,E,\sigma_1)$ for the both shown scenarios are shown in Fig. 4.27. The changes of values are marked in bold.

deploy(S,E,σ0)

	e1	e2	e3	e4
s1	M	HS	0	0
s2	M	0	0	P
s3	0	0	0	M
s4	0	0	M	0
s5	0	M	0	0
s6	0	0	M	0
s7	0	0	0	M
s8	0	0	M	0

deploy(S,E,σ1)

	e1	e2	e3	e4
s1	**P**	**M**	0	0
s2	**P**	0	0	**M**
s3	0	0	0	**P**
s4	0	0	M	0
s5	0	M	0	0
s6	0	0	**P**	0
s7	0	0	0	M
s8	0	0	M	0

Figure 4.27: Deployment matrices for scenarios σ_0 and σ_1 for the example shown in Fig. 4.23

Table 4.11 shows the content of the property $available : F \times \Sigma \rightarrow \{0,1\}$ for the two shown scenarios σ_0 and σ_1. The functional features f_3 and f_6 become unavailable in σ_1 (see also Fig. 4.26). This is okay as $failOp(f_3) = 0$ and $failOp(f_6) = 0$. Important is that $available(f_1,\sigma_1) = 1$ and $available(f_2,\sigma_1) = 1$, due to the fail-operational requirements $failOp(f_1) = 1$ and $failOp(f_2) = 1$.

Table 4.11: Availability of functional features in the two shown scenarios for the example shown in Fig. 4.23

f_i	$available(f_i,\sigma_0)$	$available(f_i,\sigma_1)$
f_1	1	1
f_2	1	1
f_3	**1**	**0**
f_4	1	1
f_5	1	1
f_6	**1**	**0**
f_7	1	1
f_8	1	1

4.6.10 Size of the Scenario Graph

If we only consider isolations of one single failing system element (one execution unit or one software component), the amount of degradation/failover scenarios in the scenario graph SG is $|\Sigma^+| = |E| + |S|$.

However, when considering a second subsequent isolation of another element due to a second failure, and when also distinguishing the chronological order of the two isolations, the amount of possible different degradation/failover scenarios becomes $|\Sigma^+| = (|E| + |S|) + ((|E| + |S|) * (|E| + |S| - 1))$.

In order to determine $|\Sigma^+|$ for chains of more than two subsequent isolations, we first consider sequences of isolations of only execution units at a first glance. Later, we add the consideration of isolations of software components.

Multiple isolations of execution units: Fig. 4.28 shows an example scenario graph SG considering sequences of isolations of executions units $e \in E$, based on an initial set of 4 execution units (|E| = 4). The labels of the nodes of the SG show here only the indexes i of the alive execution units $e_i \in EA \subseteq E$. For instance, instead of writing e_1, e_2, e_3, e_4, we write $1, 2, 3, 4$ due to limited space.

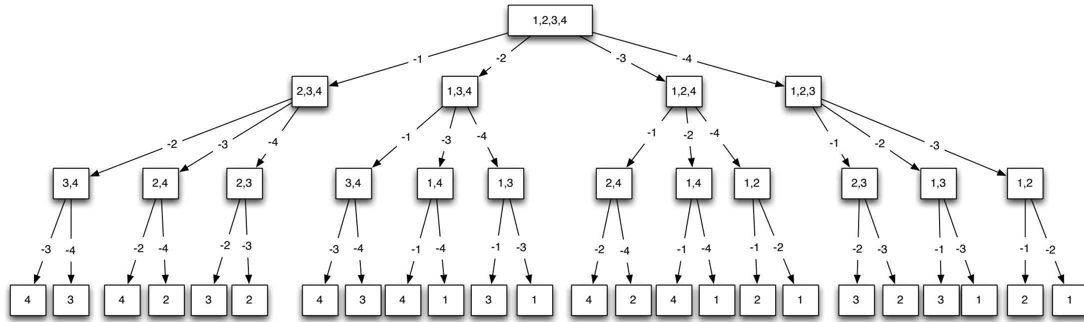


Figure 4.28: Example SG considering $n \leq 3$ isolations of 4 execution units, while considering the consecutive order of isolations

With 4 execution units, scenarios have to be considered containing 1, 2 or 3 subsequent isolations of execution units. It does not make sense to analyze a situation in which all execution units are isolated, as the result is obvious, because no single ASWC can be executed anymore and hence, no single software based functional feature can be kept available anymore. Hence, the maximum isolation sequence length of a degradation scenario is $|E| - 1$.

But also scenarios with less than $|E| - 1$ isolations have to be analyzed. Hence, with 4 execution units, overall there exist $|\Sigma^+| = |E| + (|E| * (|E| - 1)) + (|E| * (|E| - 1) * (|E| - 2)) = 4 + (4 * 3) + (4 * 3 * 2) = 40$ different degradation scenarios. Together with the root node in which all execution units are active, we obtain a number of $|\Sigma| = 41$ nodes in the graph shown in Fig. 4.28. Each node, except from the root node, represents a degradation scenario. During this approach, we distinguish different consecutive sequences of isolations, leading to identical sets of isolated and active execution units.

If we have a set of 5 execution units instead of 4 execution units, the result would be $|\Sigma^+| = |E| + (|E| * (|E| - 1)) + (|E| * (|E| - 1) * (|E| - 2)) + (|E| * (|E| - 1) * (|E| - 2) * (|E| - 3)) = 5 + (5 * 4) + (5 * 4 * 3) + (5 * 4 * 3 * 2) = 5 + 20 + 60 + 120 = 205$ different scenarios of consecutive sequences of one or more isolations due to failing execution units. Together with the initial node, the scenario graph would have $|\Sigma| = 206$ nodes. With 6 execution units, there are $|\Sigma^+| = 1236$ different degradation/failover scenarios, and with 7 execution units, there are $|\Sigma^+| = 8659$ different degradation/failover scenarios. The number of scenarios grows progressively with the number of execution units.

However, if we neglect the consecutive sequence in which the failures respectively the isolations appear, but instead analyze the sets of isolated and active execution units, the amount of scenarios drastically decreases. The SG nodes can be merged that contain identical sets of active execution units. In Fig. 4.28, all SG nodes with two active execution units exist twice, and all SG nodes with one active execution unit exist six times.

Fig. 4.29 shows the scenario graph in which duplicated nodes are merged to a single node. The structure of the scenario graph is known as *Hasse diagram* in mathematics, in this case structuring the partition of a given set (of all execution units) in at most two partitions. The first partition contains the intact execution units, the second partition contains the isolated failed execution units. The scenario graph contains only those vertexes representing a partition of intact execution units. If we analyze this graph based on all possible paths from the top root node (σ_0) to all other nodes, we obtain the same result as shown in Fig. 4.28, namely $4+4*3+4*3*2 = 40$ different paths, meaning 40 degradation scenarios.

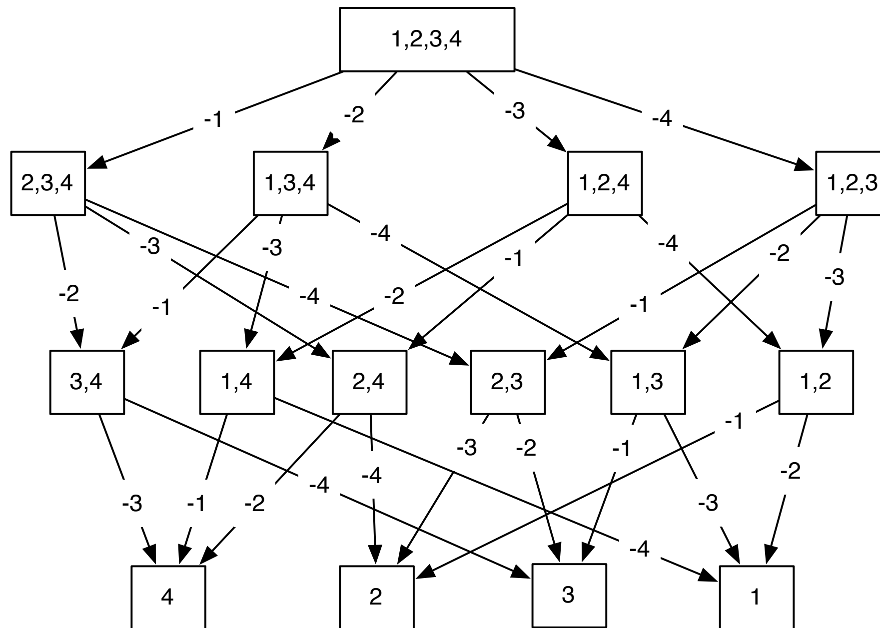


Figure 4.29: Example SG considering $n \leq 3$ isolations of 4 execution units, without considering the consecutive order of isolations

If we analyze the graph in Fig. 4.29 according to the nodes only, we analyze the sets of isolated and active execution units, neglecting the consecutive sequence of the isolations. In this way, we obtain $|\Sigma^+| = 4+6+4 = 14$ degradation scenarios. Inclusive the initial root node, the graph has $|\Sigma| = 15$ nodes.

For each set E of execution units, the amount of nodes of such a scenario graph is $2^{|E|}-1$. In case of 5 execution units, the graph has $|\Sigma| = 2^{|E|}-1 = 31$ nodes. In case of 6 execution units, the graph has $|\Sigma| = 2^{|E|}-1 = 63$ nodes.

This amount of nodes can be as well calculated by considering the amount of different combinations of active execution units, using the binomial coefficient $\binom{n}{k} = n! \,/\, (k! \cdot (n-k)!)$. For each set E of execution units, the amount of nodes is $|\Sigma| = \sum_{k=1}^{|E|} \binom{|E|}{k}$. Please notice here the difference between the set of scenarios Σ and the sum symbol $\sum$.

For 5 execution units, there are $|\Sigma| = \binom{5}{5} + \binom{5}{4} + \binom{5}{3} + \binom{5}{2} + \binom{5}{1} = 1+5+10+10+5 = 31$ nodes, see Fig. 4.30. For 6 execution units, there are $|\Sigma| = \binom{6}{6} + \binom{6}{5} + \binom{6}{4} + \binom{6}{3} + \binom{6}{2} + \binom{6}{1} = 1+6+15+20+15+6 = 63$ nodes. Always, all but one nodes represent different degradation scenarios. The initial root node represents no degradation scenario, but the initial situation with full-fledged system functionality.

Notice that for each example with $|E|$ execution units, the number of nodes is equal to the sum of the elements of the $|E|^{th}$ level of the pascal's triangle minus 1, because we do not consider the scenario in which all execution units are isolated.

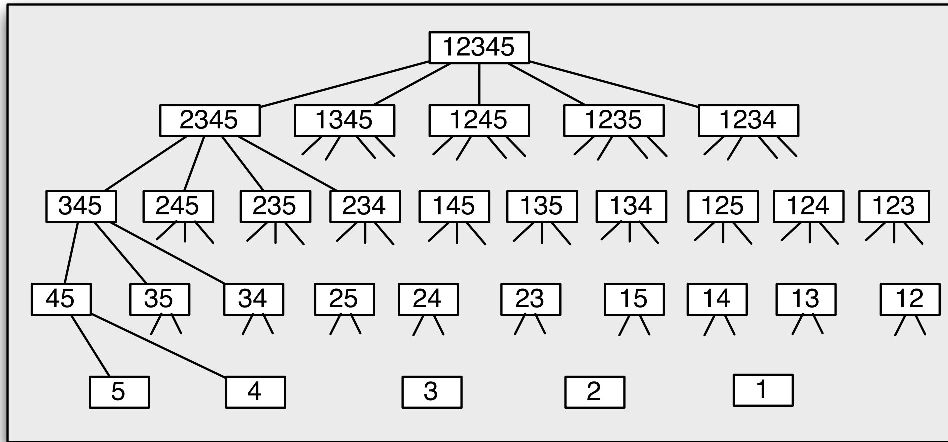


Figure 4.30: Example SG considering $n < 5$ isolations of 5 execution units, without considering the consecutive order of isolations

Multiple isolations of execution units and software components: If we add the consideration of isolations of application software components (ASWCs) to the scenario graph SG shown in Fig. 4.28 or 4.29, we obtain of course a much higher amount of nodes in SG. The nodes represent both the set of alive execution units and also the set of active software components. When considering both isolations of execution units and isolations of software components, the scenarios have a maximum sequence length of $|S| + |E| - 1$, assuming that finally all software components are isolated and all but one execution units are isolated. If we neglect the consecutive sequence of isolations, the graph has $\sum_{k=1}^{|E|+|S|} \binom{|E|+|S|}{k}$ nodes.

If desired, the analysis can be configured to the effect that only a subset of SG is analyzed automatically, and the residual graph is partially analyzed interactively, to reduce the runtime of the analysis. For instance, it may be configured that all single failure scenarios become automatically analyzed, meaning all scenarios in which exactly one execution unit or exactly one ASWC becomes isolated. All further combinations can be analyzed interactively, by specifying the subset of assumed isolated failed execution units and ASWCs.

4.7 Supporting Degradations of Single Functional Features

In this section we show how the formal model can be extended to support the description and analysis of degradations on feature level. A degradation on feature level is the substitution of a primary full-fledged functional feature by a secondary degraded functional feature, which fulfills only a subset of the functional requirements of the full-fledged feature, or potentially provides less quality of service (QoS). An example of a degradation on feature level is given by a full-fledged functional feature that provides a steer-by-wire application of a vehicle inclusive some active assistance functions (like lane-keeping or collision-avoidance), and a degraded corresponding feature that supports only rudimentary manual steering without active assistance functions. Hence, there is a difference between full fail-operational behavior (the same functional feature is kept alive after a failure), and degraded fail-operational behavior (a degraded version of the functional feature is provided after a failure).

On the software architecture level, we also consider degradations of application software components (ASWCs) by substituting a full-fledged ASWC by a degraded version of that ASWC, implementing another degraded specification. A faulty ASWC can be substituted by a degraded ASWC, contributing to realize a degraded feature version of the feature that the faulty ASWCs was realizing. One intention behind the substitutions is to establish *diversity*. Diversity is useful to handle systematic faults, either by providing two different realizations of the same functional feature, or by providing two realizations of a full-fledged and a degraded functional feature. In this thesis we address the latter case. To ensure diversity, a degraded ASWC may got implemented by another team or supplier than the full-fledged ASWC. The full-fledged and the degraded functional feature have different sets of realizing ASWCs, but the sets could intersect, meaning that some ASWCs may be reused.

A degradation of a functional feature may be required for instance because the system resources become insufficient to provide the full-fledged feature, or because an ASWC of the full-fledged feature has to be isolated due to a systematic fault leading to a fail behavior deviating from its specification. Assuming that a degraded feature requires less resources than the full-fledged feature, a feature degradation can be an adequate reaction in failure scenarios to deal with the decreasingly available resources. At runtime, a failover mechanism has to perform the substitution of the full-fledged feature by the degraded feature, while ensuring that no total service interruption of this feature combination occurs which is longer than acceptable. The runtime failover mechanism is out of scope of this thesis.

4.7.1 Assumed Design Principles for Feature Degradations

Certain design principles should be considered to limit the space of possible realization combinations in feature degradation scenarios. This mirrors in the possible degradation scenarios of the software architecture and its application software components (ASWCs). In this thesis, we suggest the following design principles and assume a compliant feature degradation design and software architecture design.

- Each functional feature is substitutable by at most one degraded functional feature, not by a set of multiple degraded functional features.
- Multiple functional features do not share the same degraded functional feature. Hence, a degraded functional feature relates to exactly one higher functional feature (partial injective degradation relationship).
- Each ASWC is substitutable by at most one degraded ASWC, not by a set of multiple degraded ASWCs.
- Multiple normal ASWCs do not share the same degraded ASWC. Hence, a degraded ASWC relates to exactly one normal ASWC (partial injective degradation relationship).

- A degraded functional feature can be further degraded, building a chain of multiple degradations.
- A degraded ASWC can be further degraded, building a chain of multiple degradations.
- Original normal and alternative degraded ASWCs might have different subscription ports, as the set of required input data might be different.
- Original normal and alternative degraded ASWCs might have different publication ports.
- If a normal ASWC contributes to realize multiple full-fledged functional features, then it has the same degraded ASWC for all of these functional features. This means, if one of the full-fledged functional features is switched to its degraded version, denoting that the normal ASWC is substituted by its degraded ASWC, then the deactivation of the normal ASWC has effect to all other full-fledged features that the normal ASWC realized. Also all these related full-fledged features have to be switched to their degraded versions, meaning that potentially more other normal ASWCs have to be substituted by their degraded ASWC version to establish the right set of active ASWCs.
- If a full-fledged functional feature is realized by multiple normal ASWCs, then the degraded functional feature may be realized with the help of a subset of the same normal ASWCs, plus some degraded ASWCs that substitute some normal ASWCs. Some additional normal ASWCs may be deactivated without substituting them with degraded ASWCs.
- The required resources (WCET, memory) of a degraded ASWC are less or equal than the required resources of the corresponding normal ASWC.

4.7.2 Extension of the Formal System Model

For some functional features $f \in F$, there may exist a degraded version of that feature. A degraded functional feature is a feature fulfilling a subset of the functional requirements of the original full-fledged feature, potentially with a worse quality of service.

Definition 18 Let $f \in F$ be an identifier of a full-fledged functional feature. Let $f' \in F$ be an identifier of a degraded version of f. We define $\mathrm{degf} : F \to F \cup \{\bot\}$ as the degradation relationship between the original full-fledged feature $f \in F$ and the degraded version $f' \in F$ of that feature, with $\bot \notin F$. With $\mathrm{degf}^{-1} : F \to F \cup \{\bot\}$ we define the inverse of degf.

$$\mathrm{degf}(f) = \begin{cases} f' \in F & \text{if } f' \text{ is the degraded version of feature } f \\ \bot & \text{if feature } f \text{ has no degraded version} \end{cases}$$

$$\mathrm{degf}^{-1}(f') = \begin{cases} f \in F & \text{if } f' \text{ is the degraded version of feature } f \\ \bot & \text{if } f' \text{ is not a degradation of another feature, but a full-fledged feature} \end{cases}$$

As listed in previous section 4.7.1, we assume that two features never share a degraded feature, meaning that $\mathrm{degf} : F \to F \cup \{\bot\}$ is an injective partial mathematical function (but not surjective, because not each element of F is a degraded feature). Because degf is injective but not surjective, also degf^{-1} is a mathematical partial function (as well injective, but not surjective).

Also chains of degradations can be expressed by this. We call the first degradation of a full-fledged feature a degradation of first order (written as f'). A further degradation of the first order degraded feature we call a degradation of second order (written as f''), etc.

Let $F^o \subseteq F$ be the subset of non-degraded full-fledged features, let $F' \subset F$ be the subset of features that are degraded features of first order, and let $F'' \subset F$ be the subset of features that are degraded features of second order (degradations of a degraded feature of first order) with $F' \cap F'' = \emptyset$, $F^o = F \setminus (F' \cup F'')$, then $\mathrm{degf} : F^o \to F' \cup \{\bot\}$ and $\mathrm{degf} : F' \to F'' \cup \{\bot\}$ are partial injective and surjective functions, but still not total functions (see Fig. 4.31).

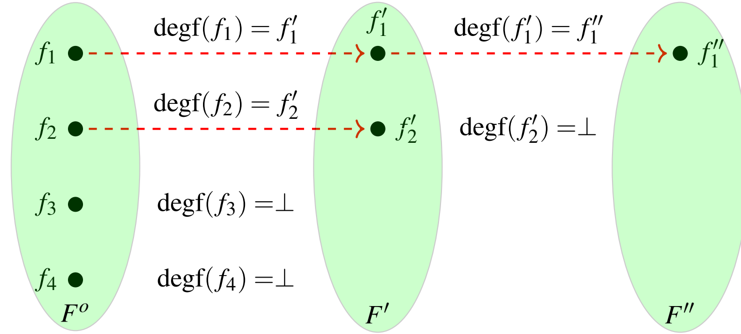


Figure 4.31: Example degradation relationships between full-fledged and degraded functional features

For some ASWCs $s \in S$, there may exist a degraded version $s' \in S$ of that ASWC. We define a degradation relationship between ASWCs as follows.

Definition 19 Let $s \in S$ be a normal ASWC, realizing a full-fledged functional feature. Let $s' \in S$ be a degraded version of s. We define $\mathrm{degs} : S \to S \cup \{\bot\}$ as the relationship between a normal ASWC $s \in S$ and its degraded version $s' \in S$, with $\bot \notin S$. With $\mathrm{degs}^{-1} : S \to S \cup \{\bot\}$ we define the inverse of degs.

$$\mathrm{degs}(s) = \begin{cases} s' \in S & \text{if } s' \text{ is the degraded version of ASWC } s \\ \bot & \text{if ASWC } s \text{ has no degraded version} \end{cases}$$

$$\mathrm{degs}^{-1}(s') = \begin{cases} s \in S & \text{if } s' \text{ is the degraded version of ASWC } s \\ \bot & \text{if } s' \text{ is not a degraded version of any other ASWC} \end{cases}$$

Fig. 4.32 shows an example of an ASWC degradation and its relation to a feature degradation. Fig. 4.33 shows another view on the same example.

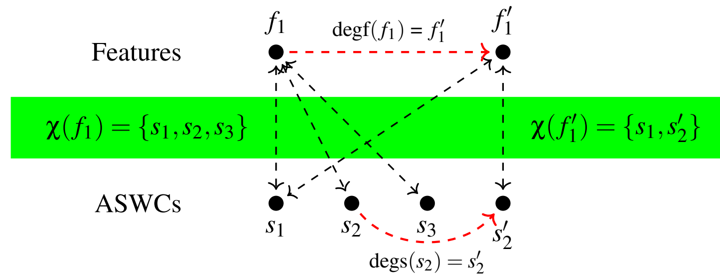


Figure 4.32: Example relation between normal and degraded ASWCs, realizing a full-fledged feature f_1 and the corresponding degraded feature f_1'

Also chains of degradations can be expressed for ASWCs, like it is the case for feature degradations.

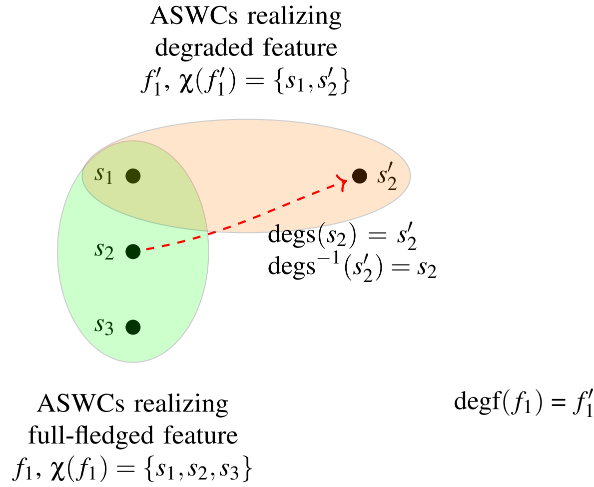


Figure 4.33: Another view on the example from Fig. 4.32, relation between normal and degraded ASWCs, realizing a full-fledged feature f_1 (green ellipse) and the corresponding degraded feature f_1' (orange ellipse)

Fig. 4.34 shows Def. 18 and 19 in context, reusing the example which was shown in Fig. 4.32, plus showing also the mapping of ASWCs to ASWC-Clusters. Also some of the predefined properties of the features and ASWCs are shown.

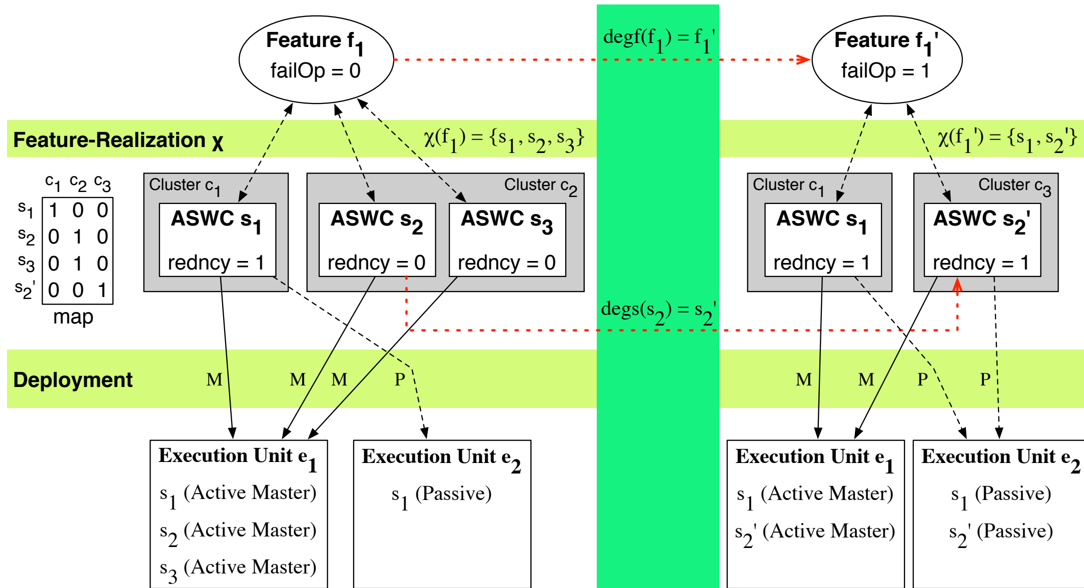


Figure 4.34: Example for the definitions w.r.t. feature degradation

The full-fledged feature f_1 in Fig. 4.34 is realized by overall three ASWCs s_1, s_2 and s_3. The three ASWCs are mapped to two different ASWC-Clusters c_1 and c_2. The three ASWCs s_1, s_2 and s_3 get deployed to the execution units e_1 and e_2, partially redundantly. Feature f_1 has a degraded version f_1', which is realized by overall two ASWCs s_1 and s_2'. The ASWC s_1 is reused from the realization of feature

f_1, and the ASWC s'_2 is a degraded version of ASWC s_2, which was realizing parts of f_1. The degradations are indicated by the horizontal red dashed arrows, labeled with $\text{degf}(f_1) = f'_1$ respectively $\text{degs}(s_2) = s'_2$.

ASWCs s_1 and s'_2 have a redundancy level of 1 and become deployed to both execution units. We assume here redundancy by cold-standby slaves. Hence, only one redundant instance is active, the other instance is passive (not executed in schedule).

Full vs. Degraded Fail-Operationality of Functional Features: As already introduced in section 4.4.2, each functional feature $f \in F$ has a property $\mathit{failOp}(f)$, defining in which sense it is required to be fail-operational. With introducing full-fledged and degraded functional features, we now distinguish between the requirement to be full fail-operational or degraded fail-operational. We express this distinction by assigning property $\mathit{failOp} : F \to \mathbb{N}_0$ to both $f \in F$ and $f' \in F$ with $\text{degf}(f) = f'$. If $\mathit{failOp}(f) > 0$, then feature $f \in F$ must be kept available with full-fledged functionality during the first $\mathit{failOp}(f)$ hardware or software failures, and is not allowed to be degraded meanwhile. More generally, it is allowed that feature f becomes degraded after $\mathit{failOp}(f) + 1$ hardware or software failures, and it is allowed that f becomes deactivated completely after $\mathit{failOp}(f') + 1$ failures, in case $\text{degf}(f') = \perp$. For instance, this means that if $\mathit{failOp}(f) = 1$ and $\mathit{failOp}(f') = 3$, then the full-fledged feature f has to survive the first failure and can be degraded to f' after a second failure. The degraded feature f' itself has to survive the third failure and can be deactivated after a fourth failure (see Fig. 4.35).

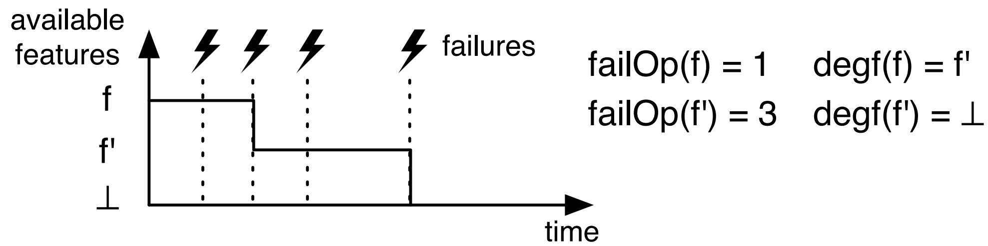


Figure 4.35: Example of a feature degradation over time

4.7.3 Extended Overview of Properties

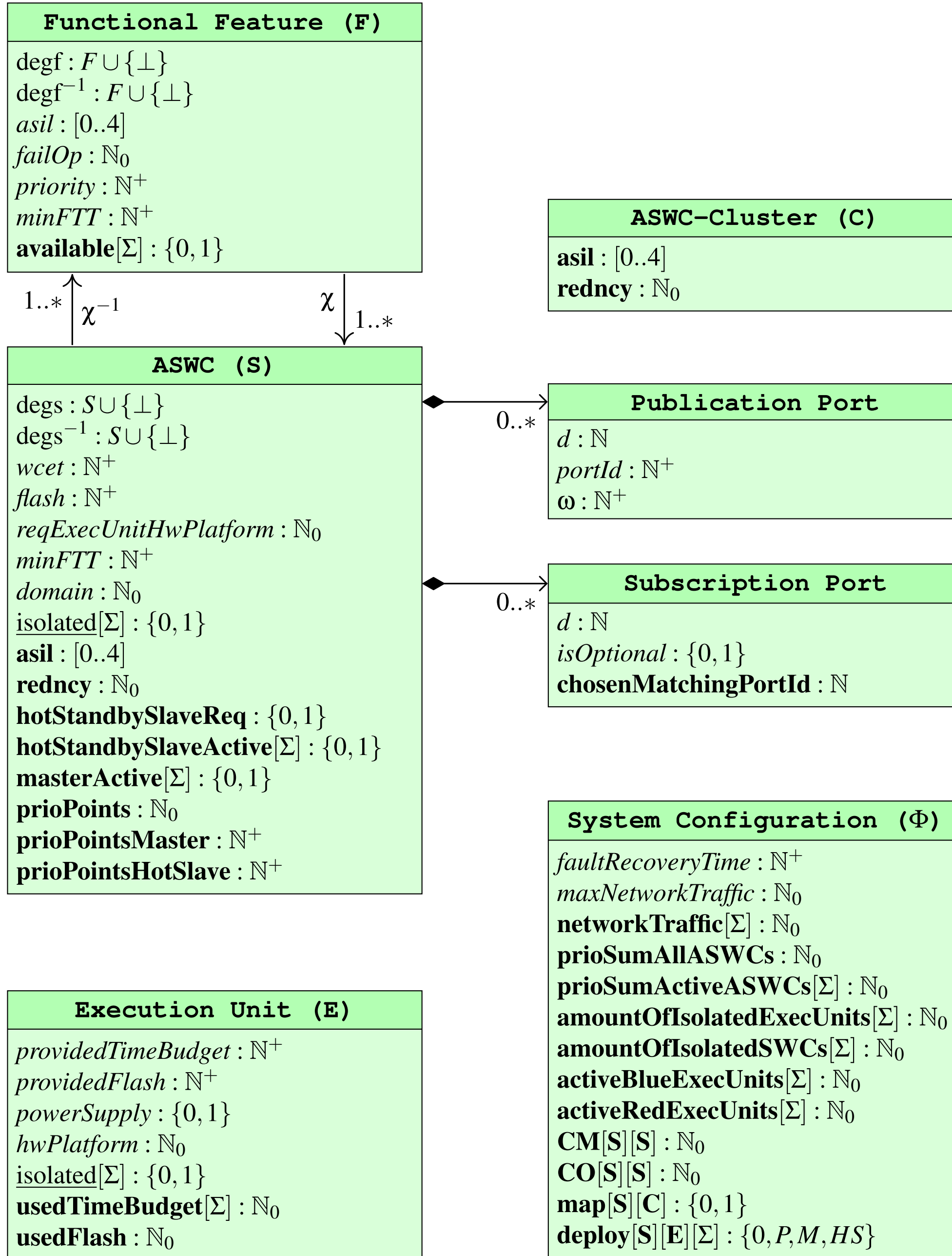


Figure 4.36: Class-diagram representation of the formal model, incl. degradations

Fig. 4.36 shows an extended version of Fig. 4.14 (shown in section 4.4.1), enriched with the degradation properties introduced in Def. 18 and Def. 19. Again, **bold** properties are solution properties (decision variables), underlined properties are variated to represent the scenarios Σ. Those properties that we

extended with the scenario parameter Σ in section 4.6.3 are shown in the class-diagram representation in Fig. 4.36 with having an array dimension $[\Sigma]$, as $|\Sigma|$ different solution values exist, one for each analyzed scenario $\sigma \in \Sigma$.

4.7.4 Formal Constraints for Feature Degradations

Requirement 10 If an ASWC s has a degraded version $s' = \text{degs}(s)$ and the related degraded feature $f' \in \chi^{-1}(s')$ has a small $minFTT(f')$, such that s' needs a hot-standby slave, there is a special requirement exception for the deployments. For the degraded ASWC s', a hot-standby slave may be active, while no master is active, as long as the full-fledged ASWC s has an active master and the full-fledged feature $f = \text{degf}^{-1}(f')$ is available.

Fig. 4.37 shows an example to illustrate requirement R_{10}.

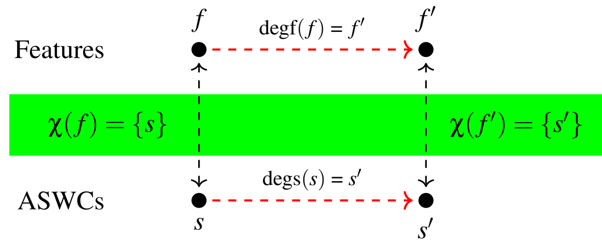


Figure 4.37: Example for requirement R_{10}

Let the features have the properties $failOp(f) = 0$ and $failOp(f') = 1$, meaning that after a failure, f is allowed to become unavailable, but f' must be available and can therewith substitute feature f in a degraded manner. The necessary redundancy levels of the ASWCs are $redncy(s) = 0$ and $redncy(s') = 1$. If $minFTT(f') \leq faultRecoveryTime$, a hot-standby slave is required for s'. In the initial deployment with $available(f) = 1$, there is now a special situation in which s' has a hot-standby slave, but no master, as f' should not be available as long as f is available. The reason for keeping the hot-standby slave of s' active is to enable a fast enough failover time from s to s' while switching from feature f to f'. This situation is expressed by constraint $C_{10.1}$.

Constraint 10.1 (R_{10}):

```
1  ∀s,s' ∈ S : ∀σ ∈ Σ:
2  Implies(                                          | if all the following is true:
3    And(degs(s) = s',                               |  s' is degraded version of s
4         degs^-1(s') = s,                           |  and vice versa
5         Or(masterActive(s,σ) = 1,                  |  master of s is active, or
6            hotStandbySlaveActive(s,σ) = 1          |  hot-slave of s is active
7         )
8    ),                                              | then:
9    And(masterActive(s',σ) = 0,                     |  master of s' should be inactive
10        If(hotStandbySlaveReq(s') = 1,             |  if hot-slave of s' is required
11           hotStandbySlaveActive(s',σ) = 1,        |  then hot-slave should be active
12           hotStandbySlaveActive(s',σ) = 0         |  else not
13        )
14   )
15 )
```

Further requirements and corresponding formal constraints exist, like a constraint describing the failover from a failing full-fledged ASWC to its degraded version. However, we do not list the full set of constraints here, as the principles in expressing the other constraints are similar.

4.7.5 Example C – Feature Degradation

In this section, we show an example of a feature set containing a feature degradation, as well as the corresponding ASWCs that realize these features. We show how this example can be degraded when execution units or ASWCs have to be isolated.

Table 4.12: Example set of functional features and realizing software components

Feature f_i	**asil**(f_i)	**failOp**(f_i)	**ASWCs** $s_i \in \chi(f_i)$	**asil**(s_i)	**rednc**y(s_i)	**flash**(s_i) in kB	**wcet**(s_i) in ms
Full-fledged Features:							
f_1 : Steer-By-Wire (with assistance)	D	0	s_1	D	1 (hot-slave)	10	1.5
			s_2	D	0	10	1
			s_3	D	0	10	1
f_2 : Parking Assistance (active)	C	0	s_3	D	0	—	—
			s_4	C	1 (cold-slave)	10	0.5
f_3 : Drive-By-Wire	D	1	s_5	D	1 (cold-slave)	10	1.3
f_4 : Infotainment	QM	0	s_6	QM	0	17	0.5
Degraded Features:							
f_1' : Steer-By-Wire (without assistance)	C	1	s_1	D	1 (hot-slave)	—	—
			s_2'	C	1 (hot-slave)	5	0.5
f_2' : Parking Assistance (passive)	C	1	s_3'	C	1 (cold-slave)	3	0.1
			s_4	C	1 (cold-slave)	—	—
f_1'' : Steer-By-Wire (limp home)	C	∞	s_2''	C	1 (hot-slave)	2	0.1

Table 4.12 shows the properties of the example. The first three columns show four full-fledged features $\{f_1, f_2, f_3, f_4\}$ and two degraded features (f_1', f_2') with $\text{degf}(f_1) = f_1'$ and $\text{degf}(f_2) = f_2'$. The right five columns show the ASWCs that realize the features. The property values, like the ASIL levels, are fictional and not related to a real case-study. Some of the ASWCs contribute to realize multiple features, like s_3 which contributes to realize features f_1 and f_2, meaning that $\chi^{-1}(s_3) = \{f_1, f_2\}$. Due to this, s_3 is shown in two rows. In rows where ASWCs are repeated, properties are printed in shade of gray, the wcet and flash are written as '—'.

Fig. 4.38 shows the realization relationship $\chi(f_i)$ between features $f_i \in F$ and ASWCs $s_i \in S$.

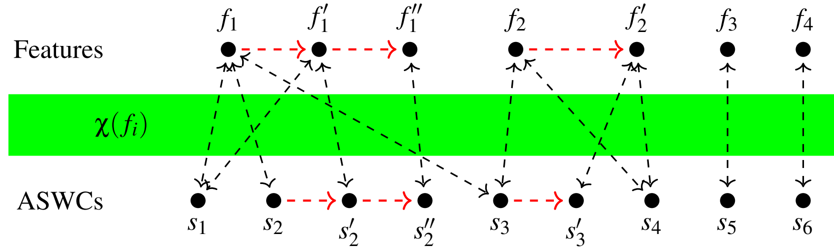


Figure 4.38: Realization relationship $\chi(f_i)$ between functional features and ASWCs for the example of Table 4.12

Fig. 4.39 shows the feature degradations of the example from a different perspective. As mentioned, ASWC s_3 contributes to realize both full-fledged functional features f_1 and f_2. Furthermore, ASWC s_1 is used to realize both the full-fledged feature f_1 (upper green ellipse) and the related degraded feature f_1' (upper orange ellipse). Hence, $\chi^{-1}(s_1) = \{f_1, f_1'\}$. The same holds for s_4 and features f_2 and f_2'. It can be seen in the figure that the example contains also ASWC degradations, namely $\text{degs}(s_2) = s_2'$ (applied during the feature degradation $\text{degf}(f_1) = f_1'$) and $\text{degs}(s_3) = s_3'$ (applied during the feature degradation $\text{degf}(f_2) = f_2'$). Features f_3 and f_4 from the example are not shown in Fig. 4.39, as these have no degraded versions.

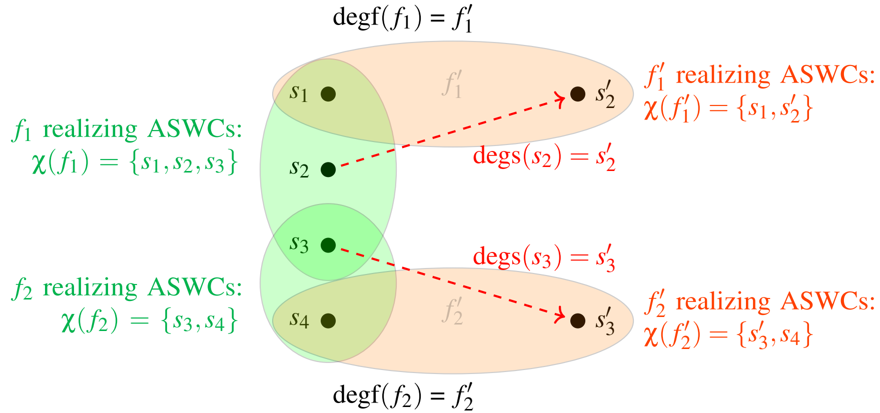


Figure 4.39: Realization of full-fledged features f_1 and f_2, as well as corresponding degraded features f_1' and f_2' by partially shared ASWCs (f_1'' not shown)

Communication channels: Fig. 4.40 lists the communication ports of the ASWCs in the example. The resulting directed communication channels between the ASWCs are shown in Fig. 4.41, as well as how the mapping of ASWCs to ASWC-Clusters is done. The clusters separate ASWCs with different ASIL, different redundancy and different hot-standby slave requirements. For instance, cluster c_1 contains contains all ASWCs which have $asil(s_i) = 4$ (ASIL D), $redncy(s_i) = 1$ and $hotStandbySlaveReq(s_i) = 1$. The ASWCs which are active as master in the initial failure-free deployment are *green*, the ASWCs which are active only as hot-standby slave but with no master in the initial deployment are *yellow*, and the ASWCs

which are inactive in the initial deployment are *white*. The latter two cases hold only for degraded ASWCs (s'_2, s'_3 and s''_2), because their full-fledged ASWCs are active as master.

ASWC	Published data items	data item weights $\omega(s_i, p_j)$	Mandatorily Subscribed data items	Optionally Subscribed data items
s_1	d_1	1	-	-
s_2	-	-	d_1, d_2	-
s_3	d_2, d_3	2, 4	-	-
s_4	-	-	-	d_3
s_5	-	-	-	-
s_6	-	-	-	-
s_2'	-	-	d_1	-
s_3'	-	-	-	-
s_2''	-	-	-	-

Figure 4.40: Publication and subscription ports of ASWCs in the example

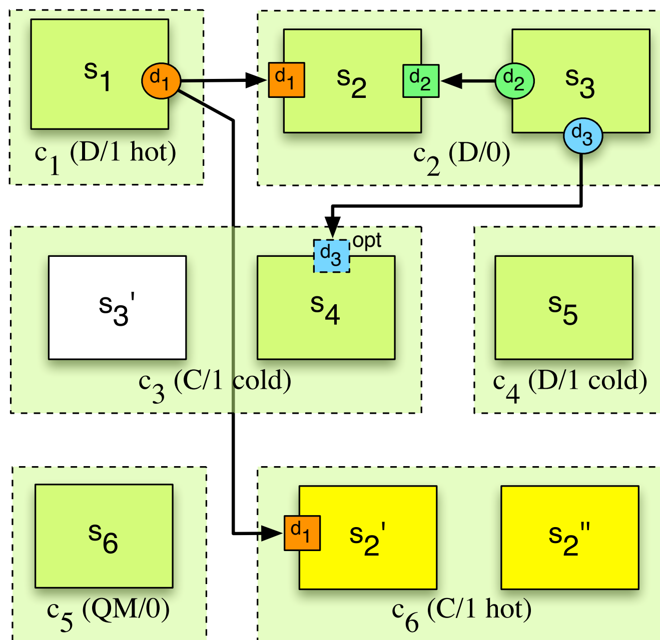
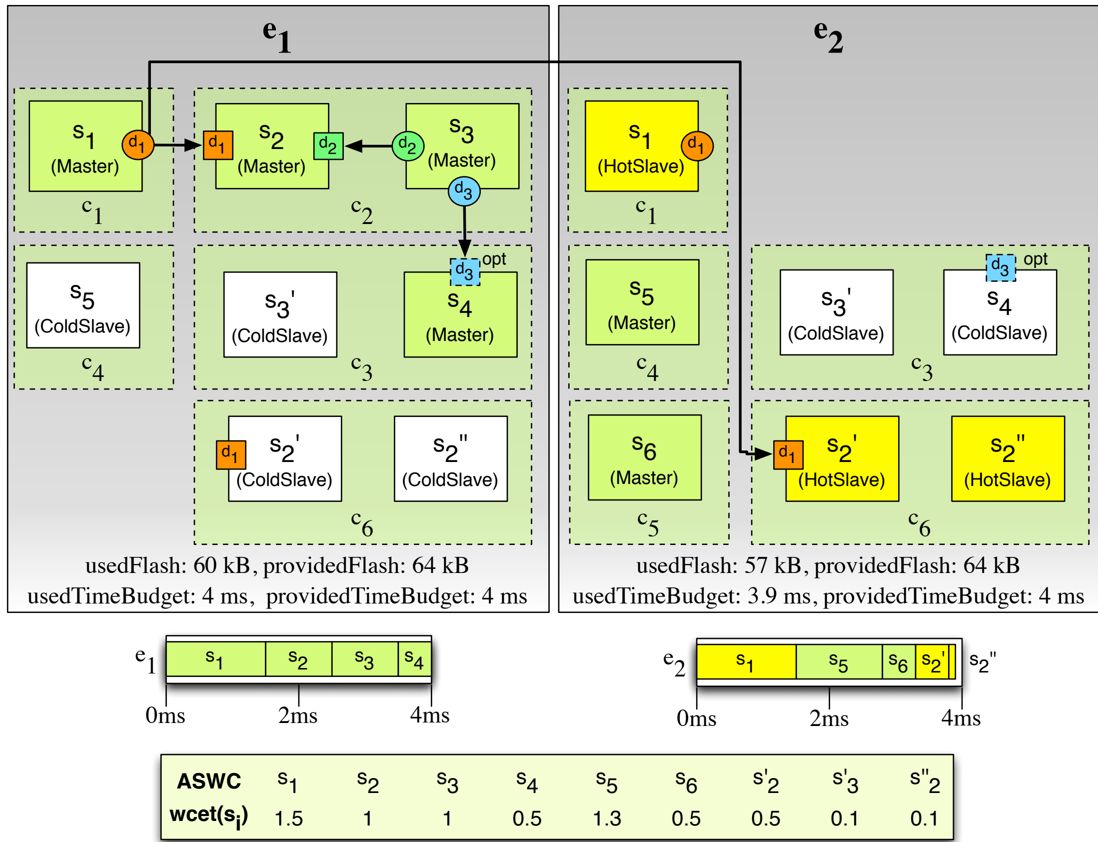


Figure 4.41: ASWC-Clusters and communication channels between ASWCs

Initial deployment solution for the example: The shown example set of ASWCs should now be deployed on two execution units $e_1, e_2 \in E$, each having a provided time budget of 4ms to execute ASWCs in each execution cycle, $providedTimeBudget(e_i) = 4ms$. Furthermore, both execution units have $providedFlash(e_i) = 64kB$ in this example.

We now consider five different failure scenarios. In scenario 1 the first execution unit e_1 has a failure and has to be isolated, with the result that no ASWCs can be executed anymore on e_1. In scenario 2 the second execution unit e_2 has to be isolated. In scenario 3 the master instance of ASWC s_1 has to be isolated, in scenario 4 the ASWC s_2 has to be isolated and in scenario 5 the ASWC s_3 has to be isolated.

We use the introduced formal model to calculate deployment solutions for these scenarios, using the Z3 SMT solver to calculate the results. Several formal constraints ensure the validity of follow-up deployments after isolations of execution units or ASWCs. We present some of these constraints in section 4.6.5. We parse the solution model returned by the SMT solver to obtain all required data needed to give feedback to the user about the analysis results.



ASWC	s_1	s_2	s_3	s_4	s_5	s_6	s'_2	s'_3	s''_2
wcet(s_i)	1.5	1	1	0.5	1.3	0.5	0.5	0.1	0.1

Figure 4.42: An initial deployment solution for the example

Fig. 4.42 shows the initial valid deployment solution for the example. Also exemplary schedules of the execution units are shown. In the schedule, it can be seen that for instance s'_2 is not executed in the initial solution, as it is a passive cold-standby slave which only becomes active if s_2 gets lost. Also the redundant cold-standby slave of s_5 on e_1 is not executed initially, as it only is a backup for the case that the master of

s_5 on e_2 gets lost. However, the components which are not executed in the schedule need flash memory space. On execution unit e_1, 55kB of flash memory are used in this example (10+10+10+10+5+10).

Also the ASWC-Clusters are shown. Five clusters are created for the given set of ASWCs. Those ASWCs are mapped to the same cluster, which have the same properties of $asil(s_i)$ and $redncy(s_i)$.

In Fig. 4.42, there are also shown some communication channels between the ASWCs. ASWC s_2 receives data from both s_1 and s_3. ASWC s_4 receives data from s_3 optionally, meaning that s_4 can also work without the input from s_3.

Tab. 4.14 shows how the solution property *prioSumAllASWCs* is calculated. The total sum of priority points is $prioSumAllASWCs = 68$. This value is composed of the following single values.

Table 4.14: Calculation of *prioSumAllASWCs*

s_i	$prioPointsMaster(s_i)$	$prioPointsHotSlave(s_i)$
s_1	7	6
s_2	6	—
s_3	6	—
s_4	6	—
s_5	7	—
s_6	2	—
s'_2	6	5
s'_3	6	—
s''_2	6	5
	$\sum 52$	$\sum 16$
	$\sum 68$	

For instance, $prioPointsHotSlave(s_1) = 4 + 1 + 1 = 6$ because it is an ASIL D component (+4), has a redundancy level of 1, and is a hot-standby slave (+1).

As the degraded ASWCs s'_2, s'_3 and s''_2 are not active as master in the initial solution, but only as hot-standby slave (if required), $prioSumActiveASWCs(\sigma_0)$ has a smaller value in the initial root scenario σ_0, as shown in Tab. 4.15.

Table 4.15: Calculation of $prioSumActiveASWCs(\sigma_0)$

s_i	$prioPointsMaster(s_i)$	$prioPointsHotSlave(s_i)$
s_1	7	6
s_2	6	—
s_3	6	—
s_4	6	—
s_5	7	—
s_6	2	—
s'_2	—	5
s'_3	—	—
s''_2	—	5
	$\sum 34$	$\sum 16$
	$\sum 50$	

The set of available functional features in the initial root scenario σ_0, meaning the set of features having solution property $available(f_1) = 1$, is $\{f_1, f_2, f_3, f_4\}$. As all full-fledged features are available, no one of the degraded features f_1', f_2' and f_1'' has to be available.

Analysis of degradations for the example: Figures 4.44, 4.45, 4.46, 4.47 and 4.48 show the follow-up deployments for the mentioned considered failure scenarios σ_1 to σ_5.

In each of these scenarios, exactly one system element fails, hence all functional features with a fail-operational level bigger or equal than 1 have still to be available, unless they are a degraded version of a full-fledged feature which is still available. In the example, the features having property $failOp(f_i) > 0$ are f_3, f_1', f_2' and f_1''. Fig. 4.43 shows the features with their *failOp* requirements, as well as the realization relationships χ between features and ASWCs.

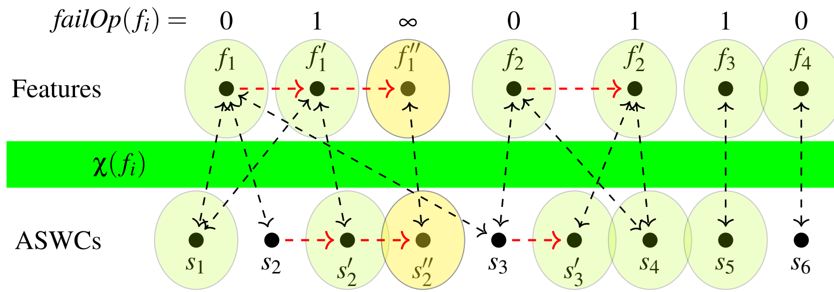


Figure 4.43: Features with fail-operational requirements (example from Table 4.12)

Due to $\chi(f_i)$, the following ASWCs are required to be active as master in all situations in which a single system element fails to keep available the features with $failOp(f_i) = 1$.

$$s_1, s_2', s_3', s_4, s_5$$

However, s_2' and s_3' are only required to be active as master, if their full-fledged versions, s_2 and s_3, are isolated or deactivated. If s_2 resp. s_3 are still active as master (meaning that features f_1 resp. f_2 are still available), then s_2' resp. s_3' are not activated as master (because f_1' resp. f_2' are not required to be available). Hence, either s_2 or s_2' and either s_3 or s_3' are active as master.

$$s_2 \vee s_2' \qquad s_3 \vee s_3'$$

The double degraded feature f_1'', which is a degraded version of f_1', has only to be available if f_1' is not available. This is not allowed in scenarios of single failures! Hence, f_1'' is not required to be available in scenarios in which only one system element fails, but later if multiple system elements fail simultaneously. This requires that s_2'' is active as hot-standby slave in all situations in which a single system element fails, to prepare for later usage as master.

$$s_2''$$

This means, only the following ASWCs are allowed to be deactivated in single failure scenarios, as they purely realize features which have no fail-operational requirement ($failOp(f_i) = 0$):

$$s_2, s_3, s_6$$

Scenario σ_1: The scenario σ_1 represents the situation in which execution unit e_1 has to be isolated, see Fig. 4.44. In σ_1, beside others, the instances of s_2 and s_3 on e_1 are lost. As s_2 and s_3 are not deployed redundantly, features f_1 and f_2 cannot be provided anymore. Hence, it is necessary to activate the degraded features f_1' and f_2', to fulfill their required level of fail-operationality. To be able to activate f_1' and f_2', the instances of s_1, s_2', s_3' and s_4 on e_2 have to become an active master. However, in order to activate the cold-standby slaves of s_3' and s_4 on e_2 to become a master, s_6 has to be deactivated on e_2, as otherwise the $providedTimeBudget(e_2)$ would be exceeded. This means, feature f_4 is lost in scenario σ_1 as well, what is okay as $failOp(f_4) = 0$. Hence, the set of available functional features in scenario σ_1 is $\{f_1', f_2', f_3\}$.

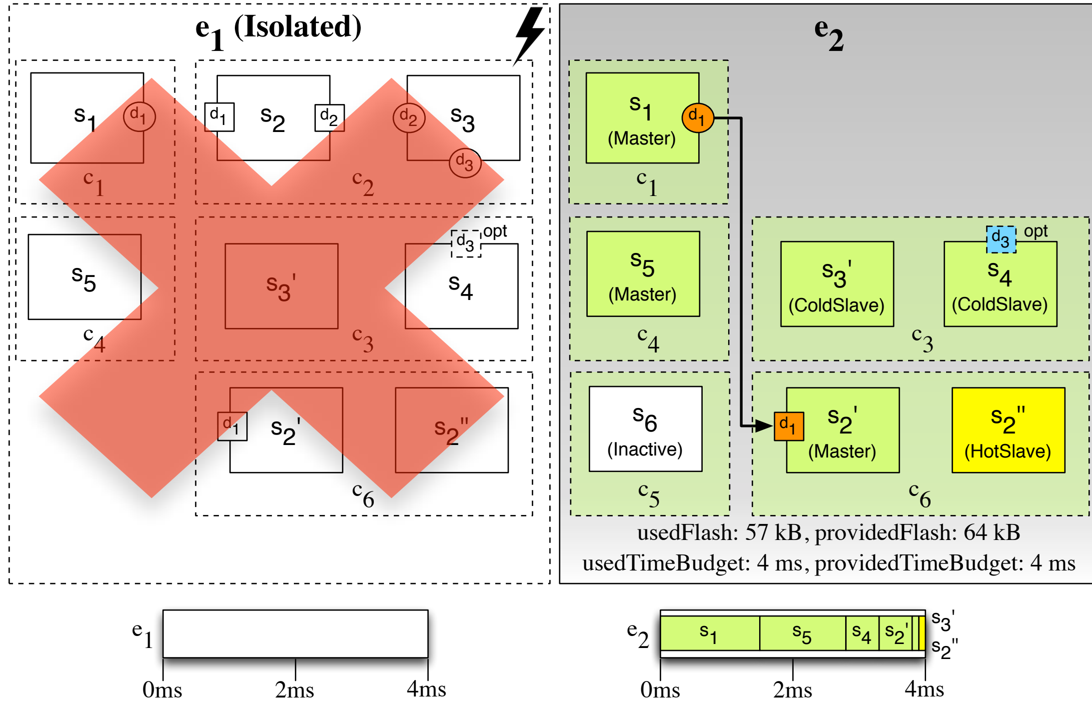


Figure 4.44: Deployment in scenario σ_1 after isolation of execution unit e_1

Tab. 4.16 shows how the value of $prioSumActiveASWCs(\sigma_1)$ is calculated.

Table 4.16: Calculation of $prioSumActiveASWCs(\sigma_1)$

s_i	$prioPointsMaster(s_i)$	$prioPointsHotSlave(s_i)$
s_1	7	—
s_4	6	—
s_5	7	—
s_2'	6	—
s_3'	6	—
s_2''	—	5
	$\sum 32$	$\sum 5$
	$\sum 37$	

Scenario σ_2: When the execution unit e_2 has to be isolated (Fig. 4.45), the cold standby-slave of s_5 on e_1 has to be activated, because s_5 realizes feature f_3 which is required to behave fully fail-operational. In order to be able to activate s_5 on e_1, some other ASWCs have to be deactivated on e_1. However, deactivating s_1 would cause the loss of f_1 and f'_1. Deactivating s_4 would cause the loss of f_2 and f'_2. Hence, this is not allowed. Thus, s_2 and s_3 have to be deactivated to free enough space in the schedule to be able to activate s_5. Hence, s'_2 has also to be activated on e_1 in order to be able to provide feature f'_1. Feature f_1 cannot be provided anymore in this scenario. Also f_2 cannot be provided anymore as s_3 is inactive, but f'_2 can be provided because s_4 can operate standalone without the optional input.

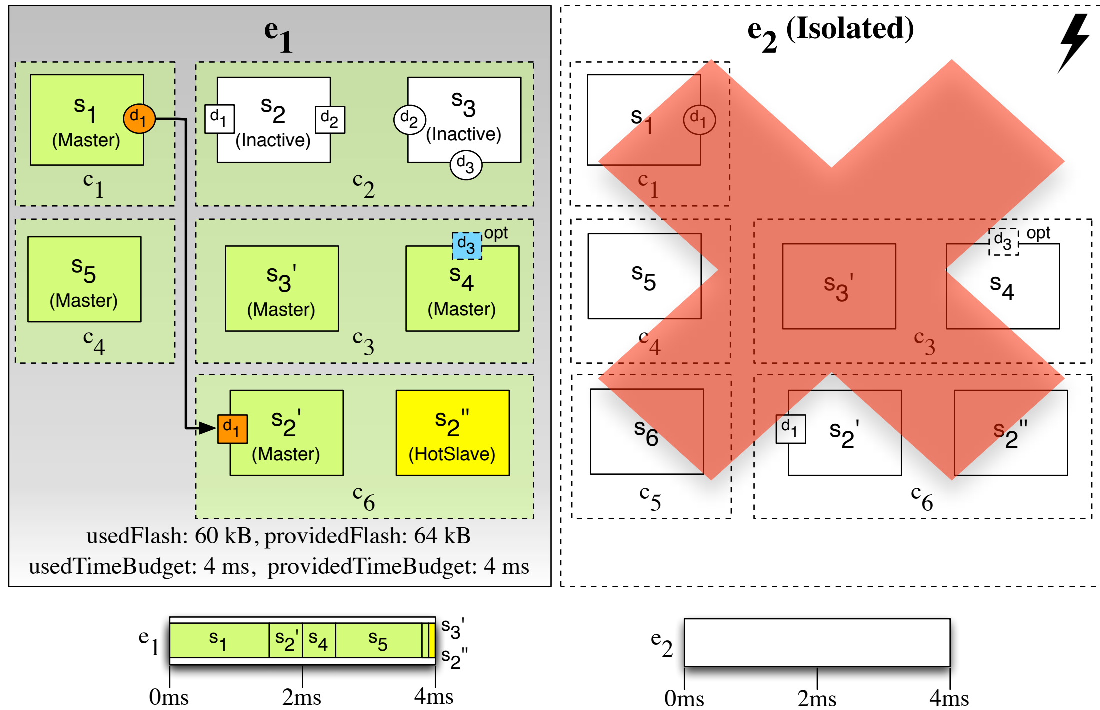


Figure 4.45: Deployment in scenario σ_2 after isolation of execution unit e_2

Scenario σ_2 is very similar to scenario σ_1. The set of active ASWC instances in σ_2 and σ_1 are identical, and also the set of available features are identical in both scenarios. The sum of priority points is $prioSumActiveASWCs(\sigma_2) = 37$, as well. The set of available functional features in scenario σ_2 is $\{f'_1, f'_2, f_3\}$.

The solution property $usedTimeBudget(e_i, \sigma_j)$ changes in the degradation scenarios, as the sets of active ASWC instances change, an thereby also the schedules of the execution units. However, the solution property $usedFlash(e_i)$ keeps unchanged, as the binary of deactivated or isolated ASWCs is assumed to be kept stored in the flash memory.

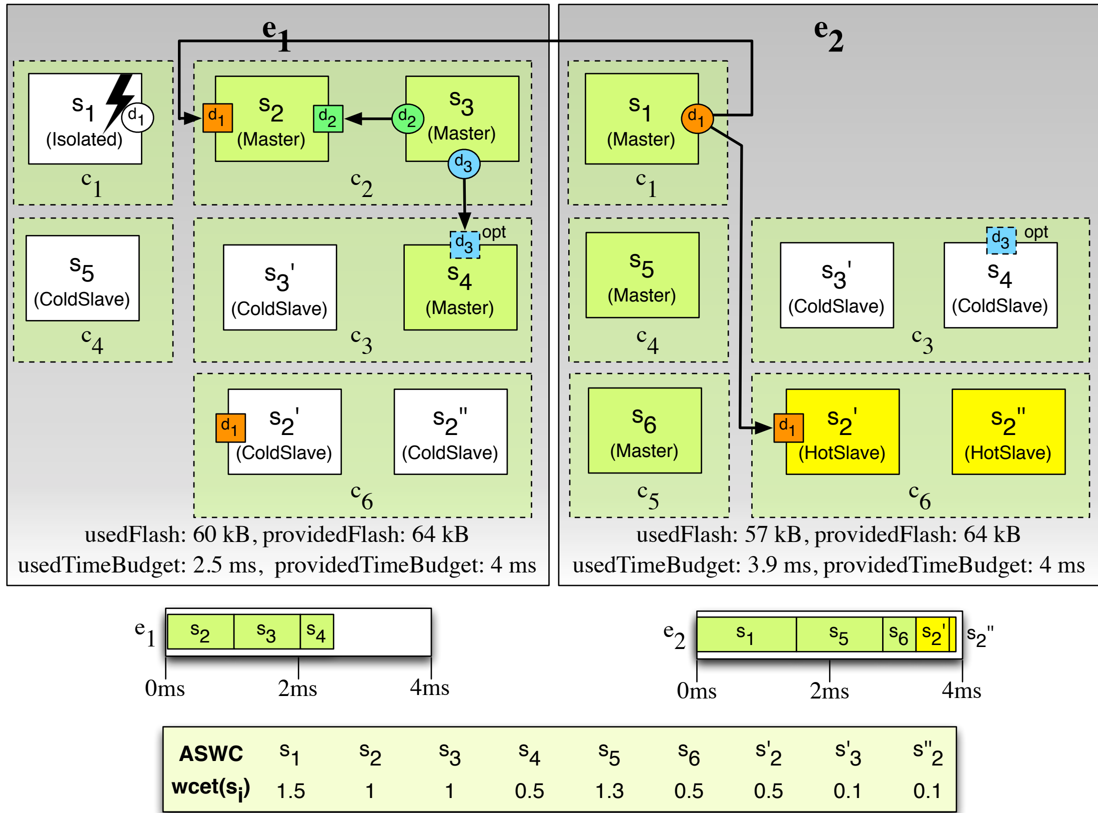
Scenario σ_3: In case a failure is detected in the master instance of ASWC s_1 on execution unit e_1, there are three options.

1. The master instance can be just restarted. This might be appropriate under the assumption that very unlikely environment conditions lead to transient unconsidered inputs to s_1 causing the detected error, assuming that these environment conditions will not occur again after the restart. But a cold-restart from the initial state of s_1 is not appropriate, as a hot-restart is required for this component to ensure its $minFTT(s_1)$. An alternative would be to use a *checkpoint mechanism* [274] to be able to provide a warm-restart based on the last valid saved state of s_1. We do not further consider this, as no deployment changes are necessary for this. Also a hardware problem might have caused an internal miscalculation of s_1, based on correct inputs. But as described before, such a hardware problem can be detected by using dual-modular redundant (DMR) execution units with bitwise comparison between the two lanes, assuming that never the identical hardware problem will arise in both lanes, leading to exactly the same miscalculation of s_1 on both lanes.
2. The master instance of s_1 can be isolated and the hot-standby slave of s_1 on e_2 becomes the new master (see Fig. 4.46). This is only appropriate under the assumption that the detected error is not caused by a systematic fault in the code of s_1 (like a typical bug), which would also be present in the hot-standby slave instance, based on the identical code. Instead, the detected error should be caused by some external reason, that makes it preferable to not just restart s_1 on the same execution unit.
3. Assuming a systematic fault in the code of s_1, both the master and the hot-standby slave instance should be isolated, because also the hot-standby slave would be erroneous. However, for the given example this results in an unsatisfied constraint representing the fail-operational requirements, because feature f_1 and also its degraded version f_1' would become unavailable, as s_1 contributes to both feature versions. Hence, this solution is invalid for the given example, as $failOp(f_1') = 1$ is violated. To resolve this problem, a dissimilar alternative implementation of s_1 should be introduced into the architecture. This may be done either by adding a component s_{1b} implementing the same specification as s_1 and hence realizing also functional feature f_1, or otherwise by providing a degraded version s_1' of s_1, which is able to realize the degraded version f_1' of f_1. In the latter case, the software architecture of the example has to be changed such that $s_1 \notin \chi(f_1')$, but $\chi(f_1') = \{s_1', s_2'\}$. Like s_2', also s_1' has to have the property $redncy(s_1') = 1$. Furthermore, when s_1 is isolated, also s_2 is lost due to the mandatory subscription of data item d_1. As also s_2' subscribes d_1 mandatorily, the additional degraded ASWC s_1' has to publish d_1 to obtain a valid solution. All this can be checked with our approach.

Fig. 4.46 shows the result for scenario σ_3 under the assumption that the 2nd option is preferred, namely a failover in form of switching the hot-standby slave of s_1 on e_2 to become the new master. We encoded the formal constraints, which represent valid degradation scenarios, such that this option is chosen. However, the constraints can be modified such that one of the other options can be analyzed. The set of available functional features in scenario σ_3 is $\{f_1, f_2, f_3, f_4\}$.

Scenario σ_4: When s_2 has to be isolated (Fig. 4.47), then feature f_1 cannot be provided anymore. ASWC s_2' has to be activated to provide the degraded f_1'. ASWC s_3 is kept active to continue providing feature f_2. The set of available functional features in scenario σ_4 is $\{f_1', f_2, f_3, f_4\}$.

Scenario σ_5: When s_3 has to be isolated (Fig. 4.48), also s_2 has to be deactivated as s_2 needs mandatory data from s_3. Hence, features f_1 and f_2 cannot be provided anymore. But the degraded features f_1' and f_2' can be provided, as s_1, s_2' and s_4 are active. The set of available functional features in scenario σ_5 is $\{f_1', f_2', f_3, f_4\}$.



ASWC	s1	s2	s3	s4	s5	s6	s'2	s'3	s''2
wcet(si)	1.5	1	1	0.5	1.3	0.5	0.5	0.1	0.1

Figure 4.46: Deployment in σ_3 after isolation of master of s_1 and failover to its hot-standby slave

Hence, all requirements w.r.t. to full and degraded fail-operational behavior can be fulfilled in the scenarios σ_1, σ_2, σ_4 and σ_5. In scenario σ_3, it depends on whether a failover from the master to a hot-standby slave makes sense in the considered fault model. If yes, also σ_3 is valid. If not, a degraded version s_1' of s_1 should be added to the software architecture, which publishes data item d_1 and is able to realize the degraded functional feature f_1' in cooperation with s_2'.

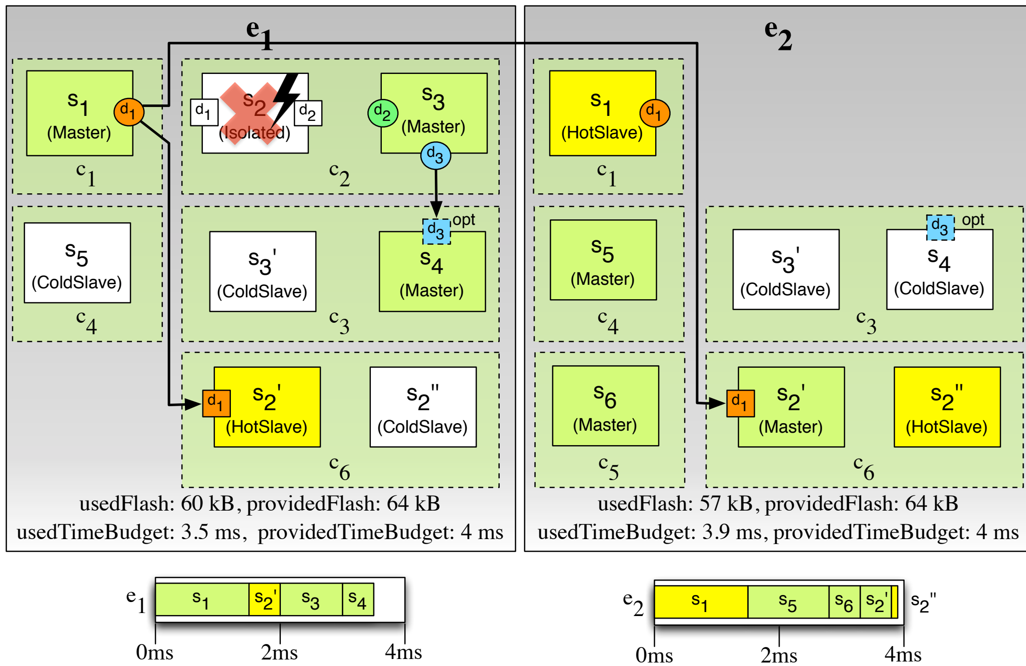


Figure 4.47: Deployment in scenario σ_4 after isolation of s_2

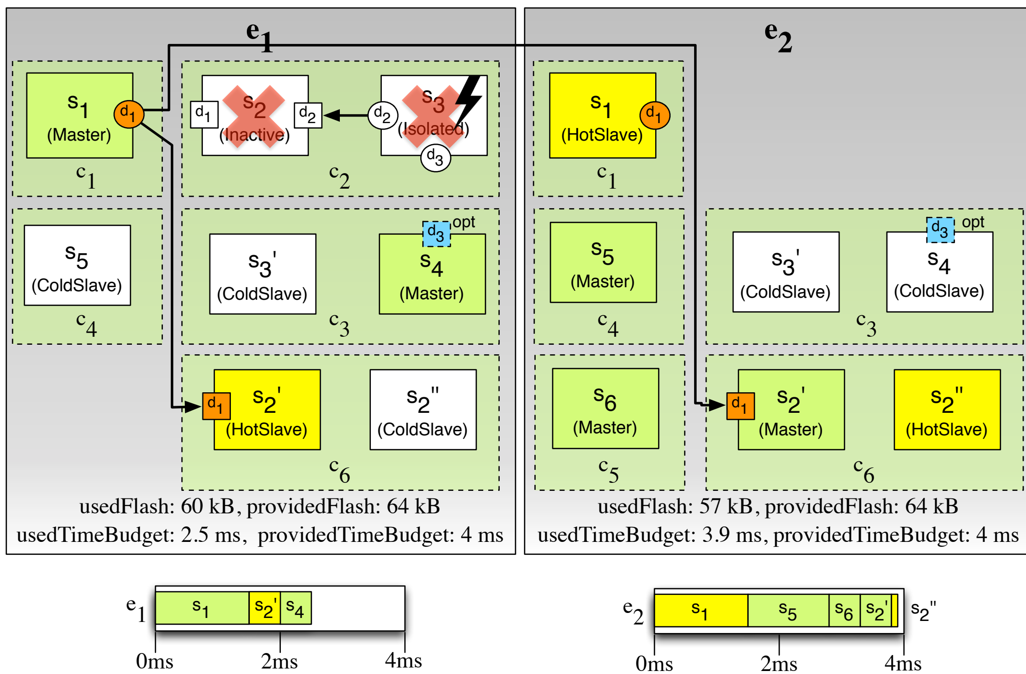


Figure 4.48: Deployment in scenario σ_5 after isolation of s_3

4.8 Formalization of Optimization Objectives

In order to obtain the desired analysis results for failover and degradation scenarios, we define a number of optimization objectives for the Z3 SMT solver. These objectives ensure, for instance, that the level of degradation in the scenarios does not become higher than necessary, resulting in a graceful degradation. The objectives also ensure that the correct system elements are set to be isolated in the scenarios, and that some aspects of the conducted deployment synthesis are optimized. This opens a multi-objective optimization problem design space.

Before introducing the constraints in more detail, we need to take care about how conflicting constraints shall be resolved. As we define multiple objectives that are partially contradicting, we need to define how the solver shall handle the conflicts. The Z3 SMT solver offers three alternatives:

1. *lexicographic* prioritization, considering the objectives with decreasing importance in the order in which the objectives are added to the model

2. *pareto*, returning subsequently multiple pareto efficient points on the pareto frontier of the problem for each call of the check() function of the solver

3. *box*, returning the optimal value for each objective independently from the other objectives.

For our analysis, we need to establish a fixed prioritization of the objectives. This can be reached by resolving conflicting objectives with the lexicographic order, configurable in Z3 Python API by command

```
1 s = Optimize()
2 s.set(priority='lex')
```

That means that the objectives are prioritized in order of their appearance, with decreasing priorities. This equals to the following code in SMT-LIB 2.x format [27], according to the documentation of the optimization facility of the Z3 SMT solver. [4]

```
1 (set-option :opt.priority lex)
```

We now introduce the set of optimization objectives that we use during our analysis to obtain a solution that is optimal with respect to the addressed characteristics.

- Objective O_1: minimize the amount of isolated execution units and ASWCs, to avoid undesired additional isolations in the solution. The objective prohibits the solver to set more elements to be isolated than desired. So, only those elements are isolated, for which the isolation property is explicitly set to 1 in the constraint that controls the setup of degradation scenarios.

- Objective O_2: maximize the amount of available full-fledged features that have degraded versions, to avoid undesired switches to degraded features. For instance, if a degraded feature is realized by more ASWCs than the full-fledged feature, having a higher sum of priority points, without O_2 the following objective O_3 would potentially cause an undesired switch from the full-fledged feature to the degraded feature.

- Objective O_3: maximize value of property *prioSumActiveASWCs* in each degradation scenario. By this, it is reached that in case of insufficient execution resources, low critical ASWCs are deactivated first, reaching that low critical functional features become non available first.

[4] http://rise4fun.com/Z3Opt/tutorial/guide, last access May 23th 2016

- Objective O_4 is an addition to O_3. On the component level (O_3) there might be the choice that one of two components with ASIL=QM can be deactivated. Objective O_4 now ensures that the component is deactivated that realizes less functional features. Hence, O_4 defines that in general, as many features shall be kept available as possible.
- Objective O_5: minimize the network traffic. This minimization can be reached by deploying the master instances of communicating ASWCs to the same execution unit to use local communication.
- Objective O_6: maximize the amount of hold soft constraints, as introduced in section 4.6.7.

Objective 1 :

```
1 ∀σ ∈ Σ:
2 minimize(amountOfIsolatedExecUnits(σ))
3 minimize(amountOfIsolatedSWCs(σ))
```

Objective 2 :

```
1  ∀σ ∈ Σ:
2  maximize(
3        sum_{f∈F} (
4              If (  And(                              | if all the following is true:
5                        available(f,σ) = 1 ,          | f is available
6                        degf^{-1}(f) = ⊥ ,            | f is not a degraded feature itself
7                        degf(f) ≠ ⊥                   | f has a degraded version
8                    ),
9                    1,                                | then add 1 to the sum
10                   0                                 | else add 0 to the sum
11             )
12       )
13 )
```

Objective 3 :

```
1 ∀σ ∈ Σ:
2 maximize(prioSumActiveASWCs(σ))
```

Objective 4 :

```
1 ∀σ ∈ Σ:
2 maximize(sum_{f∈F}(available(f,σ)))
```

Objective 5 :

```
1 ∀σ ∈ Σ:
2 minimize(networkTraffic(σ))
```

Objective 6 :

```
1 ∀σ ∈ Σ:
2 maximize( sum_{i∈ℕ} (If(softConstraintTracker(i,σ) = 1,1,0)) )
```

If we would remove the objectives, we would still obtain valid degradation scenarios that do not violate fail-operational requirements, as these are ensured separately by hard constraints, like $C_{8.1}$ and $C_{8.2}$. But the results would be potentially inefficient, having a higher degree of degradation than necessary.

A subset of the objectives ensures to optimize the conducted deployment synthesis. From the shown objectives, this is the case for O_5. This subset of objectives might be changed, exchanged or extended, if the user has other priorities with respect to the conducted deployment synthesis. However, the location in the sequence of objectives must be carefully chosen. For instance, another comparable objective to O_5 would be to minimize the amount of required execution units (not shown). This objective would be required to be defined between O_2 and O_3, such that the added *minimize amount of execution units* objective is more important than the contradicting constraint to maximize the amount of priority points of active ASWCs (O_3).

4.9 Assumptions and Aspects that are out of Scope

4.9.1 Out of scope

The presented approach tackles a structural analysis of software architectures in certain situations that may appear after failures in the system have been detected.

However, there are certain aspects of analysis which we do not tackle in the presented approach and which we want to clarify here. Our approach has:

- no notion of time stream and time intervals
- no notion of when in time failures appear or how probable failures are
- no notion of behavior and states of features and SWCs

Hence, when we determine the availability of functional features in different degradation scenarios, we express in a boolean $\{0,1\}$ form if a functional feature can be kept available in that scenario. Classical *availability* metrics consider a period in time and give a statement about the percentage of time in this time period, for which a feature is available (see section 2.1.3). Similarly, classical *reliability* metrics use a mean-time-between-failure (MTBF) metric to calculate the probability that a system will fail after a certain period of time, under an assumed error rate, without considering healing by repair times as done for availability metrics.

Furthermore, we do not consider error detection mechanisms, separation mechanisms, isolation mechanisms for erroneous system elements, and concrete runtime failover mechanisms to activate redundant backups of lost software components. We assume that the systems runtime environment (RTE) (resp. middleware) offers appropriate techniques to do this. For instance, the RTE shall be able to perform failovers without service interruption of an affected functional feature, or if allowed with an acceptable degree of interruption.

4.9.2 Assumed properties of the system under analysis

Our approach rests upon assumptions for the system under analysis. We assume the presence of a runtime environment (RTE), comprising the ensuing concepts.

We assume that the execution units of the system are either partially homogeneous, or that the RTE of the system offers an abstraction such that software components can be deployed to execution units of different types. We assume that all sensor data and other data can be accessed from all execution units and that actuators can be controlled from all execution units. All this allows flexibility in the deployment of software components to execution units.

As scheduling policy, we assume the concept of logical execution times (LET), meaning that the software components are executed within fixed *cycles*, see Fig. 2.13. Each execution unit provides a budget of time per cycle that can be used to execute application software components (ASWCs). At start of each cycle, an image of all input data should be created, then the application software components are executed based on the input image and write output data to an output image. After all application software components finished, the output image should be transfered to the receivers, like actuators. This allows flexibility in the sequence of execution of software components and about local and remote communication. Hence, precedence relations between ASWCs have not to be considered in the schedule of the ASWCs. For simplicity, we assume that all software components are executed in every execution cycle (single-rate scheduling). Also for simplicity, we assume that ASWCs have a single operation executed by the scheduler (single task/runnable/entry-point per component). We assume that the worst-case execution time (WCET) of the scheduled ASWC-operation is identical on all execution units. Our model could be extended to support multiple runnables per component, which may have different execution periods (multi-rate scheduling), like occurring in AUTOSAR, but this is out of scope of this thesis.

To establish the communication, we assume that the execution units, sensors and actuators, are connected to each other by a reliable network topology. Hence, we do not consider failures within the network communication. As communication delay, we assume that all data between the connected devices can be transfered over the network from one cycle to the next cycle, such that all data sent in cycle x is available on all other devices at cycle x+1. This supports flexibility in the deployment, without having to analyze effects of the deployment to the end-to-end latencies of communicated data.

The Runtime-Environment (RTE) (or the Middleware) of the system under analysis has to have the following capabilities. It detects runtime failures of certain system elements, like sensors, actuators, execution units, communication buses, or software components. It isolates failing system elements from the residual system to avoid failure propagation and harm. Mixed criticality software components are separated in a temporal and spatial manner. Failover mechanisms allow to activate redundant backup components, if required. No dynamic migrations of code or binaries of software components between execution units are performed at system runtime. Degradation mechanisms handle insufficient resources after isolations. Functional features can be deactivated to become unavailable in an appropriate safe way. All this is required to establish fault-tolerance and to support fail-operational features in connection with needed degradations or deactivations of other non fail-operational features. This empowers the RTE to compensate the loss of isolated failed system elements. In this thesis, we consider isolations of execution units and software components. With the help of our approach introduced in this thesis, degradation scenarios for the described type of systems can be formally analyzed at their design time.

The principles of a system platform and corresponding RTE that conforms to our assumptions are outlined in section 2.5 and introduced in [19] [318] [36].

4.10 Explanations and Discussions about the Formal System Model

In this section we provide some additional insights and discussions about the design of our formal system model, as well as alternatives about parts of it.

4.10.1 Functional Features

Relation of feature set F and feature hierarchies, modeled as a tree: The functional features of a system are often modeled as a *feature tree* (also called *function hierarchy* [340]), expressing hierarchical compositions of features by sub-features. Fig. 4.49 shows an example feature tree on the left side. The example root feature f_{root} is composed of two sub-features, which are again composed by sub-sub-features. The leaf features are atomic and not further composed. In that context, functional features are often called *functions* [340].

We consider the feature set F, which we defined in Def. 1, as a subset of the features of such a feature tree, constructed by a horizontal slice through the feature tree. Hence, we do not consider feature compositions in the set F (see right side of Fig. 4.49).

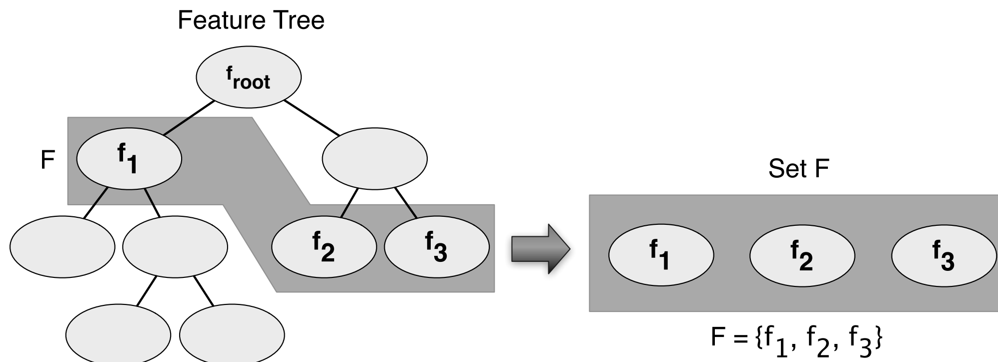


Figure 4.49: A hierarchical feature tree (left) and a set of functional features F (right) that is created based on the feature tree

In the formal analysis approach introduced in this thesis, we only consider the features of set F. Sub-features of a feature $f \in F$ are out of scope (for instance the sub-features of f_1 in Fig. 4.49). With our approach, it is possible to analyze in which situations f_1 has to be deactivated, but more fine-grained deactivations of the sub-features of f_1 are not considered. Due to this, the set F may contain most probably all leaf-features of a feature tree. But we do not restrict this here. The decision is open to the user of our approach.

Notice that we do not consider feature models, which describe the variability of software product lines (SPLs), introduced in Feature-Oriented Domain Analysis (FODA) [192] and also treated for instance in [222] [43] [42] [101]. Instead, we consider a feature tree as a specific product variant instance of a SPL feature model, and create our feature set by doing a slice through this feature tree. When extending our approach towards the consideration of feature models with variability, the feature part of our formal model (see Def. 1) has to be extended, as well as new constraints (see section 4.5.1) are required that express the feature model and valid feature configurations. Finally, multiple feature sets as concrete products of a feature model could be analyzed in one single analysis, together with analyzing the adequacy of the different associated realizing software architectures (see section 4.2.3). This is out of scope of this thesis.

Feature Dependencies: Interactions or dependencies between functional features denote how single functional features influence each other when composing the entire system behavior. In our formal model, we do not incorporate explicitly dependencies or interactions between functional features. The reason is that we do not need to describe these dependencies explicitly on feature level to perform the failure effect analysis introduced in this thesis. Instead, we perform the analysis on software architecture level. In the software architecture, the feature dependencies can be realized mainly in two ways:

1. Communication channels between software components that realize different features. In this thesis, we distinguish optional and mandatory communication channels between software components, representing optional or mandatory feature dependencies. A *mandatory* feature dependency is typically given if a feature f_i is based on another feature f_j such that f_i needs f_j to fulfill its own service. An *optional* feature dependency is given if a feature f_i is influenced somehow by another feature f_j, but f_i can also provide its service standalone.

2. Software components that contribute to realize multiple features. Depending on the implementation of the shared ASWCs, desired or undesired feature dependencies can be realized, both either in one direction, or also bidirectional. It may also be the case that a shared ASWC does not result in a feature dependency at all. We do not go deeper into detail about how such a shared usage of an ASWC results in different types of feature dependencies in this thesis for the sake of simplicity. Such dependencies are for instance further considered in [342] [340].

4.10.2 Software Architecture

Consideration of software components: Usually, when constructing a software architecture, different application software component types are modeled, and these application software component types are reused and instantiated multiple times when composing the application software architecture of the system. In this thesis, we assume that if a software component type is reused in multiple places in the software architecture, we treat these as multiple different ASWCs $s_i, s_j \in S$ in our formal input model. For simplicity, we assume that all $s \in S$ have the same component type.

However, ASWCs $s \in S$ might become deployed redundantly to multiple execution units. We call this the deployed *instances* of an ASWC. Only one of these redundant instances is acting as the so called active *master* instance, which provides the functionality of that component to the architecture.

Beside the introduced ASWCs, which are located at the application layer, other software components might exist in a system design, like SWCs located at the runtime environment (RTE) layer or at the basic software (BSW) layer. However, in this thesis, we consider only ASWCs.

Compositions of software components: In the presented formal model, we treat software components as black boxes, without considering internal structure, states or behavior. This also implies that we do not model hierarchical compositions of software components. If the real software architecture of the system under analysis applies compositions, what is usually the case for many systems, anyhow the real software architecture design should have some information about which components build a deployable unit. These deployable components shall be used as input set for our analysis. If no such information exists, the user of our approach has to select a set of software components adequately.

Selection of channels between ASWCs: In section 4.2.2 (Def. 6) we introduced that we model ports of ASWCs, but treat channels not between these ports, but aggregate the channels to be between ASWCs. Furthermore, in section 4.4.3 (see property *chosenMatchingPortId*) we introduced that we synthesize the set of used channels out of a set of channel candidates. Why do we do this?

Although we consider channels between ASWCs, we define ports of ASWCs in the input problem model to describe possible communication channel candidates between ASWCs. The benefit of selecting used channels from a bigger set of channel candidates during the analysis is that we can synthesize an optimal architecture, for instance minimizing the network traffic, if multiple candidates for channels exist, e. g., because two or more matching publications exist for one subscription. This means, we do not assume that the channels are predefined in the input problem model, but this can be supported easily, disabling the above mentioned benefit.

There are different options to control the decision about which publication port is chosen out of a set of possible matching ports to get connected to a considered subscription port. One possibility is to choose one of those whose ASWC is deployed to the same execution unit than the ASWC of the subscription port. This reduces the network traffic due to preferring local communication. The channel selection introduced in this thesis is done once for the whole analysis. This means, the channel selection is fixed and does not change in case of a transition from the initial deployment to a degraded deployment. If a selected publisher ASWC of a channel has to be deactivated in a degradation scenario, no reconfiguration is done to select another matching publisher ASWC out of the remaining set of matching publisher ports to establish a new channel to substitute the lost channel. If such a behavior is desired for a system under analysis, the formal constraint model can be adapted as future work to cover such scenarios.

Establishment of channels at runtime: We assume that the system under analysis has a runtime environment (RTE) with a module (e. g., a Broker) that technically establishes the communication channels between the ports of ASWCs. To follow the result of the analysis provided by our approach, this should be done according to the synthesized channel selection. Fig. 4.50 shows such a situation, where the ports of two software components s_1 and s_2 are interconnected by a Broker of the RTE. Such a Broker is for instance present in the XME RTE [71]. Fig. 4.50 also shows a possible white-box view into the internal structure of s_1 and s_2, showing how they might be composed by internally connected subcomponents. As mentioned, we do not consider such internal compositions in our formal system model.

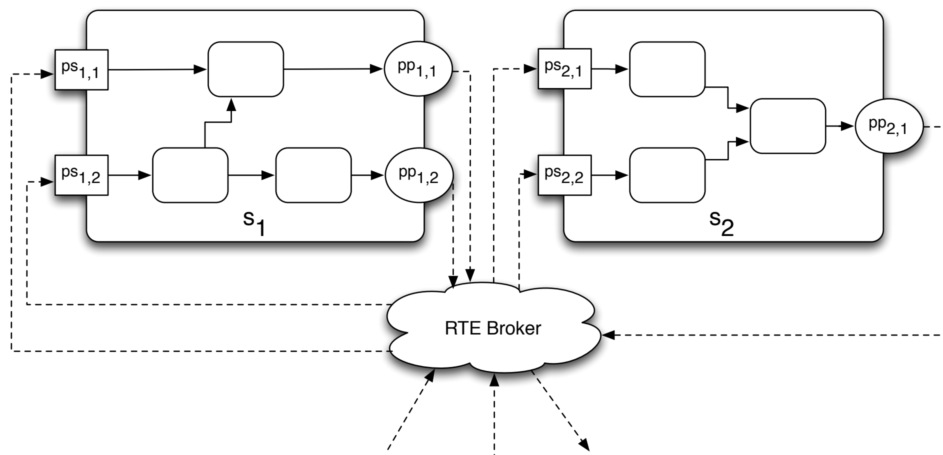


Figure 4.50: RTE Broker that connects ports by channels

Causality Relations: For the sake of simplicity, we assume in our model that each subscription of a component $s \in S$ influences each publication of s. This could be refined by specifying in detail which subscriptions influence which publication, like for instance considered as *causality-relations* in [308] or *inner port dependency traces* in [169], allowing a more fine grained analysis about failure propagations inside components and hereby which subset of publications could potentially still be provided after a subset of the subscriptions become unavailable or have insufficient quality. However, in the model introduced in this thesis, we assume that no publication can be provided at all if one mandatory subscription becomes unavailable, and all publications can be provided if some optional subscriptions become unavailable.

No service degradation in case of missing optional input data: We assume that ASWCs can provide their full service, even if optional input data is missing. Hence, no service degradation happens when optional input data is missing. A support of service degradations in case of missing optional input would be an extension point to our approach. An operating mode model would be required for ASWCs to capture different degraded operating modes of ASWCs in case of decreasing optional inputs. This could be also extended to capture missing mandatory inputs. To be able to analyze in detail which missing input has which degradation effect in the operating mode model, the channel matrices have to be redefined to be over the number of ports, not over the number of ASWCs. This is out of scope of this thesis and future work.

In section 4.7, we introduce ASWC degradations in case of missing mandatory input data by switching to a degraded version of an ASWC, which does not mandatorily rely on the missing input. This is done by deactivating the ASWC that misses mandatory input data and activating a degraded version of that ASWC which does not require this input mandatorily. The benefit is that this supports *diversity* of ASWCs.

ASWCs with different operating modes: In this thesis, we do not consider different operating modes of ASWCs, which might also be used to establish graceful degradation. If the ASWCs of the system under analysis would have different operating modes, the introduced analysis approach would not consider them. If a failed mode-based ASWC becomes isolated from the residual system, none of its operating modes can be provided anymore. Hence, in this situations, deactivations of operating modes should be considered instead of deactivations of complete ASWCs. However, this is out of scope of the thesis. As we focus on structural analysis and not on behavioral analysis, we do not model operating modes of ASWCs, but instead we model substitutions of an ASWC by a degraded version of that ASWC, as introduced in section 4.7. This allows to model degradations by switching between two ASWCs that have a degradation relationship and allows to incorporate *diversity*, as the two ASWCs could be provided by two different development teams.

Single Rate Scheduling with single entry-point: We assume in this thesis that all ASWCs are executed with the same rate in every execution cycle (single rate scheduling). We define software components to have a single entry-point operation (alias function) [5], which becomes executed by a scheduler periodically in fixed execution cycles. This means, in AUTOSAR terminology [22], each ASWC has exactly one Runnable [6] and all Runnables have the same execution rate (time-triggered period). We do not consider ASWCs with multiple executable operations in the formal model for simplicity, but the introduced formal model could be extended in this direction. With such an extension, also software components with multiple executable operations (each having an own WCET) could be supported, which are executed in periodic cycles with homogeneous or heterogeneous rates in a time-triggered manner, or even triggered by aperiodic events. Such an extension would be required to support AUTOSAR models [22], having multiple *Runnables* of ASWCs with different execution periods.

[5] see "Entry Point" on page 37 of AUTOSAR_TR_Glossary.pdf [22]

[6] see "Runnable Entity" on page 73 of AUTOSAR_TR_Glossary.pdf [22]

Why do we limit the redundancy by the amount of execution units? In section 4.4.3, we introduced the $rednc y(s)$ property and that we define this to be $rednc y(s) = min\Big(|E| - 1, max\big(failOp(f) \mid f \in \chi^{-1}(s)\big)\Big)$, for $s \in S$. We would like to explain here, why we limit the redundancy by the amount of execution units. During our proposed analysis, we consider failures of execution units and ASWCs. We interpret the fail-operational requirement of a functional feature as the amount of failures (and subsequent isolations) of execution units and ASWCs that the feature has to survive. Hence, if we would derive the redundancy for instance by $rednc y(s) = max\big(failOp(f) \mid f \in \chi^{-1}(s)\big)$, we wound obtain a problem. If we would have for instance 4 execution units and 10 ASWCs and have a feature $f \in F$ with $failOp(f) = 6$, the required redundancy of an $s \in \chi(f)$ would be $rednc y(s) = 6$, meaning that s has to be deployed 7 times. If we restrict the deployment that at most one instance of an ASWC can be deployed to each execution unit, then this redundancy level is not possible, as there are only 4 deployment targets. The question is, does it make sense to relax this restriction and allow multiple instances of the same ASWC on the same execution unit?

In case of failures of execution units, all instances get lost that are deployed to an isolated unit. In this case there is no benefit of multiple deployments onto the same unit. In case of a failure of the considered ASWC, detectable by an interface behavior violating the specified behavior, we need a closer look onto the fault that caused the failure. If an ASWC master instance failed due to a permanent software fault (like a software bug), then also the other redundant instances of that ASWC would have the same bug. Hence, starting a cold-standby slave would be identical to just restarting the master from its initial state, independently from where the cold-slave is deployed to. A failover to a hot-standby slave is not possible, as the hot-slaves behavior would have also failed. Hence, in case of a software bug, multiple local instances have no benefit. Restarting the master may be an option, if the situation leading to the occurrence of the bug is very unlikely. Otherwise, we highly recommend to introduce diversity and switch to an alternative implementation of that ASWC. In section 4.7, we consider such diversity, but combined with degraded behavior of the diverse ASWC. If the deviation from the specified behavior occurred due to a electromagnetic event leading to a bit flip in the memory of the ASWC on the execution unit (not to a bug in the ASWC), then also restarting the ASWC from its initial state would have the same effect than starting a second cold-standby slave instance of that ASWC on the same execution unit. However, in this case it may make sense to start a redundant instance of that ASWC on another execution unit, assuming that the electromagnetic event is less likely there. If the ASWC has a required hot-standby slave, then the hot-slave would anyway be always on another execution unit than the master, to avoid that master and hot-slave disappear simultaneously in case of an isolation of one execution unit.

However, in a very degraded state, in which only one single execution unit is left, it makes sense that master and hot-slave are active on the same execution unit, but only if the fault is not a permanent software bug. In case of a bug in the ASWC, both the master and the hot-slave would violate their specified behavior. In case of a electromagnetic caused failure of the master, a failover to the local hot-slave makes sense, assuming that the data of the slave was not affected. The benefit is, that the internal state of the hot-slave is already identical to the specified valid state of the master. If we would just restart the master, the drawback would be that the ASWC is restarted to its initial state. This may result in a dangerous behavior on functional feature level. Hence, only in the last sketched scenario it would make sense to deploy at most two instances of an ASWC (master and hot-standby slave) to the same execution unit. However, instead of isolating the master and failover to the local hot-slave, the same situation could be closely resolved by a *roll-back* of the master to its last valid state, under the drawback that the calculation done in the faulty execution cycle is lost and a state transfer might be deferred by one cycle. Also this can result in a dangerous behavior on feature level. But if the execution units are lockstep units with dual-modular redundant execution lanes, such an electromagnetic caused ASWC failure would probably appear only in one lane. Hence, the failed ASWC instance could be repaired by copying the correct memory image from the other lane. Such a mechanism is called *roll-forward* recovery [275] [206]. Due to these recovery

options, we do not model deployments of multiple instances of an ASWC to the same execution unit and limit the redundancy of ASWCs to $rednc y(s) \leq |E|$.

Calculation of the property prioPoints of ASWCs: There is a fact to notice about the shown calculation of the property $prioPoints : S \rightarrow \mathbb{N}_0$, which builds the basis for the sum elements leading to the value of property *prioSumActiveASWCs*, both introduced in section 4.4.4. We assume that only ASIL C or ASIL D features can have a fail-operational requirement. The reason is that lower critical features can be deactivated with little danger for severe harm. However, if we have two features, one ASIL C feature f_1 with $failOp(f_1) = 1$, and one ASIL D feature f_2 with $failOp(f_2) = 0$, the question is which feature has to be available longer? Regardless of whether this makes sense from the perspective of safety and reliability argumentation, we want our approach to be able to analyze this.

We consider the fail-operational requirement as more important than the ASIL. Hence, if a feature has to be disabled, f_2 has to be disabled first, such that f_1 satisfies its fail-operational requirement. Anyhow, a formal constraint over our model ensures that the fail-operational requirement of features must not be violated, avoiding a too early deactivation of f_1. However, in order to be able to analyze incorrect architectures with respect to which architecture elements cannot be guaranteed to hold their requirements, the constraint is a soft constraint, meaning that it can be violated in the solution. The violated constraint allows to trace back to the related ASWC and/or functional feature, allowing to give feedback to the user.

The soft constraint avoids that the SMT solver simply returns *unsat*, denoting that no valid solution for the problem can be found. Such an *unsat* gives no clue about the reason that forces the model to be unsatisfied. We introduce the soft constraints in more detail in section 4.6.7.

Hence, also the maximization of *prioSumActiveASWCs* has to take care that f_2 is disabled first. If now $s_1 = \chi(f_1)$ and $s_2 = \chi(f_2)$, it is important that s_2 becomes deactivated first before s_1, as otherwise f_1 would become unavailable first, violating its fail-operational requirement. Hence, it must hold that $prioPoints(s_1) \geq prioPoints(s_2)$. The introduced valuation adheres to this, as $prioPoints(s_1) = prioPoints(s_2) = 4$. As the values are equal, the solver has a choice. In order to maximize property *prioSumActiveASWCs*, the solver decides based on another constraint or objective about which ASWC to deactivate first. There is another optimization objective defining that the amount of violated soft constraints has to be minimized. Hence, the solver decides to deactivate s_2 first, as otherwise to fail-operational constraint of f_1 would be violated.

Alternative functional representation of map: Alternatively to the relational matrix representation *map*, which we introduced in section 4.2.2 and use in our formal system model, the mapping could also be described in a functional representation that maps each ASWC $s \in S$ to exactly one cluster $c \in C$.

$$\alpha : S \rightarrow C \text{ with } \forall s \in S : \alpha(s) = c \in C \text{ iff. } map(s,c) = 1 \tag{4.3}$$

The inverse mapping, giving the subset of ASWCs that are mapped to a given cluster, can be defined as

$$\alpha^{-1} : C \rightarrow \mathcal{P}(S) \text{ with } \forall c \in C : \alpha^{-1}(c) = \{s \in S \mid \alpha(s) = c\} \tag{4.4}$$

Notice that although α^{-1} maps an element of the domain C to potentially multiple elements of S, α^{-1} is still a mathematical function as the codomain is a set of sets, namely the power-set $\mathcal{P}(S)$ of S, and α^{-1} maps an element of the domain C to exactly one element of the codomain $\mathcal{P}(S)$. For $c_i \neq c_j$, it holds that $(\alpha^{-1}(c_i) \cap \alpha^{-1}(c_j) = \emptyset)$ and $\bigcup_{c \in C} \alpha^{-1}(c) = S$.

The functional representation allows to slenderly express constraints like $\forall s \in \alpha^{-1}(c) : asil(c) = asil(s)$, defining that all ASWCs within an ASWC-Cluster have to have an identical ASIL and that the ASIL of the cluster is derived by the ASIL of the ASWCs mapped to this cluster. However, instead of using the functional representation α and α^{-1}, we use the relational matrix representation *map* in the formal system

model to express the mapping of ASWCs to ASWC-Clusters, as the matrix representation is well suited to express all formal constraints over the elements.

4.10.3 Feature Realization

Either atomic leaf features, or also non-leaf features of a feature-tree (see Fig. 4.49) may be considered to get realized by software components of set S. This expresses the decision on which granularity level features of the feature tree are realized by a set of software components. The slice through the feature tree might be done in different ways. If a non-leaf feature is contained in F, like the left most feature f_1 of set F in Fig. 4.49, we assume that the set $\chi(f_1)$ of ASWCs realizing this non-leaf feature is the sum of the ASWCs which realize the sub-features of f_1.

4.10.4 Hardware Architecture

We assume that all execution units are connected to each other by a reliable network topology. A reliable Ethernet ring based network topology for future electric vehicles has for instance been introduced in [19].

4.10.5 System Configuration

Single instances of ASWCs per execution unit: To be able to model the deployment by the matrix $deploy : S \times E \rightarrow \{0, P, M, HS\}$, we restrict the deployment such that at most one instance of an ASWC becomes deployed to each execution unit. Hence, there are never multiple redundant instances of an ASWC on the same execution unit. If a consideration of deployments of multiple redundant instances of ASWCs to the same execution unit would be required, the deployment matrix can be extended to a deployment cube $deploy : S \times E \times m \rightarrow \{0, P, M, HS\}$, with $m \in \mathbb{N}^+$ being the maximum amount of redundant ASWC instances per execution unit.

No deployments of ASWC-Clusters: In this thesis, we consider the deployment of ASWCs onto execution units, not the deployment of ASWC-Clusters onto execution units (like it was introduced in [38] and [40]). We consider the deployment of ASWCs, because the activity of the ASWCs within a cluster may become different when single ASWCs within a cluster fail and have to be isolated. Hence, some ASWCs within a cluster may become isolated due to detected failures of that ASWCs, or they may become passivated due to insufficient resources, while other ASWCs within the same cluster are still active. Isolating the whole cluster in case of a failure of a single ASWC of that cluster is not appropriate, as potentially many other intact ASWCs would get lost and due to this, and also the realized functional features. Hence, the ASWCs within a cluster might have different activity, meaning that in one cluster instance on an execution unit, some of the contained ASWCs may be passive, while others may be active masters or hot-standby slaves. Thus, the deployment is described on ASWC level, not on cluster level.

Alternative functional representation of deploy : Alternatively to the relational matrix representation *deploy*, which we introduced in section 4.2.5 and use in our formal system model, the deployment could also be described in a functional representation that deploys each ASWC to a subset of the execution units.

$$\delta_P : S \rightarrow \mathcal{P}(E) \text{ with } \forall s \in S : \delta_P(s) = \{e \in E \mid deploy(s,e) = P\} \tag{4.5}$$

$$\delta_M : S \rightarrow \mathcal{P}(E) \text{ with } \forall s \in S : \delta_M(s) = \{e \in E \mid deploy(s,e) = M\} \tag{4.6}$$

$$\delta_S : S \rightarrow \mathcal{P}(E) \text{ with } \forall s \in S : \delta_S(s) = \{e \in E \mid deploy(s,e) = HS\} \tag{4.7}$$

The co-domain of δ_M and δ_S is a power set instead of a single value, as the sets might become empty for instance in degradation scenarios and not every ASWC requires a redundant hot-standby slave.

We further introduce $\delta_A(s) = \delta_M(s) \cup \delta_S(s)$ to express the subset of execution units to which $s \in S$ is deployed *actively*, either as master or hot-standby slave.

$$\delta_A : S \to \mathcal{P}(E) \text{ with } \forall s \in S : \delta_A(s) = \{e \in E \mid deploy(s,e) \in \{M, HS\}\} \tag{4.8}$$

Finally, we introduce $\delta(s) = \delta_P(s) \cup \delta_M(s) \cup \delta_S(s)$ to express the subset of execution units to which $s \in S$ is deployed either passively (only in memory) or actively (in memory and in schedule).

$$\delta : S \to \mathcal{P}(E) \text{ with } \forall s \in S : \delta(s) = \{e \in E \mid deploy(s,e) \in \{P, M, HS\}\} \tag{4.9}$$

However, instead of using the functional representations δ_P, δ_M, δ_S, δ_A and δ, we use the relational matrix representation *deploy* in the formal system model to express the different kinds of deployments of ASWCs to execution units. We use the relational representation because it is well suited to express the formal constraints over the elements. In the formal constraints, it is for instance required to count to how many execution units an ASWC is deployed in one of the above introduced deployment types (passively, actively as master, etc.). The relational matrix representation is well suited to offer this summations by calculating conditional sums over rows or columns of the matrix, as presented in the formal constraints listed in section 4.5.1.

Minimization of level of degradation in the considered failure scenarios: We obtain a minimization of required degradations in the analyzed scenarios by creating a concept of priorities of software components. These priorities are automatically derived by the ASIL and fail-operational levels. We calculate a sum of these priorities and formalize an objective to maximize the sum in order to minimize degradation. See property *prioSumActiveASWCs* in sections 4.4.4, 4.6.5 and 4.8. However, even if this sum would make no sense, for instance if the software architecture would contain an ASIL A component with fail-operational requirement, as well as an ASIL D component without fail-operational requirement, resulting in a fail-operational component having a lower priority than a non-fail-operational component, the resulting deployments will never become invalid. Fail-operational requirements never become violated, because their fulfillment is enforced by hard constraints. Hence, even in this case the ASIL D component would be deactivated first, even if it has a higher priority value.

CHAPTER 5 Evaluation

In this chapter, we evaluate the contributions of the approach being introduced in chapter 4. To evaluate the introduced approach, we already discussed three self constructed examples in sections 4.6.8, 4.6.9 and 4.7.5, showing the type of systems and scenarios that can be analyzed with our approach. Supplementary, in section 5.1 we discuss to what extent our research questions, presented in section 1.3, are resolved by the presented analysis approach. Further assumptions and limitations regarding our analysis are discussed in section 5.2. In section 5.3 we discuss the threats to the validity of the presented approach.

Contents

5.1 Discussion of Research Questions 153
5.2 Limitations of the presented approach 155
5.3 Threats to Validity 157

5.1 Discussion of Research Questions

In section 1.3 we introduced the following research questions (RQs) as motivation for this thesis.

RQ1: How to automatically calculate valid deployments of software to hardware, supporting the fulfillment of fail-operational requirements?

RQ2: How to formally analyze the ability to keep functional features available in scenarios of failing system elements, and decide about necessary degradations of the available feature set, incorporating necessary failovers to ensure fail-operational requirements?

RQ3: How to formalize a given system design concept and the requirements related to the safety and fault-tolerance concept of that system to be able to apply the formal analysis to this type of system?

We now discuss which parts of this thesis contribute to which research questions and to which extend we resolved these research questions.

RQ1: How to automatically calculate valid deployments of software to hardware, supporting the fulfillment of fail-operational requirements?

In sections 4.2 and 4.4, we described a formal model of deployments of software components onto hardware execution units, supporting mixed levels of redundancy and different states of redundantly deployed component instances. This allows to express constraints for the validity of synthesized deployments, as introduced in section 4.5.1. To be able to express and ensure fail-operational requirements, the formal model also contains a notion of functional features attached with required levels of fail-operationality, as well as relationships about which functional features become realized by which software components. Based on this information, we ensure that an appropriate level of redundancy of software components is built into the synthesized deployments to take care that the fail-operational requirements of the related functional features can be fulfilled by means of using redundant backup software components to replace lost primary components if necessary. We also support *mixed-criticality* by treating different criticality

levels (e. g., ASIL, but also DAL could be used). Furthermore, by means of mapping components with identical criticality to so called clusters, we synthesize deployments preparing to separate the different clusters by using a separation kernel. Combined with RQ2, we ensure that we synthesize an initial deployment that is an appropriate starting condition to fulfill all fail-operational requirements in all failure scenarios considered in RQ2, if such a deployment exists. In sections 4.6.8 and 4.6.9, we have shown the applicability of the presented approach based on two examples.

RQ2: How to formally analyze the ability to keep functional features available in scenarios of failing system elements, and decide about necessary degradations of the available feature set, incorporating necessary failovers to ensure fail-operational requirements?

To analyze the effect of internal failures of system elements and subsequent isolations of those elements, potentially leading to insufficient system resources to provide the full set of functional features, we synthesize strategies to deactivate low critical functional features in order to be able to keep high critical features with fail-operational requirements available.

However, we not only model a boolean fail-operational requirement, but we support to model different levels of fail-operationality, meaning how many subsequent failures of different internal system elements shall be survived. This is taken into account when setting up the constraints for the required levels of redundancies of software components for the deployment synthesis.

Based on assigning so called *priority points* to software components, calculated automatically based on the required safety integrity level and required level of fail-operationality, we model an objective that if software components are needed to be deactivated due to insufficient resources, then those components become deactivated which result in deactivation of functional features with as low safety integrity level as possible. Hence, in the automotive domain, those functional features with QM level are deactivated first, then ASIL A features, and so on, while always taking care that fail-operational requirements are fulfilled.

We introduced the basic concept in section 4.6 and applied it to two examples in sections 4.6.8 and 4.6.9. Furthermore, in section 4.7 we extended our model and analysis to incorporate also degradations of single functional features, empowered by supporting to model degraded versions of software components, also allowing to incorporate *diversity*. We applied this extension onto a third example in section 4.7.5.

To structure the analysis, we modeled a scenario graph and a formal model that allows to analyze the whole graph at once with one single execution of the employed SMT solver. Alternatively, also partial graphs can be analyzed. As described in section 4.6.4 and published in [38], it would be also possible to analyze each graph node separately and put an algorithm on top to adapt certain constraints during traversing through the graph to take previous results into account, like predecessor deployments. However, the *one-big-solution* approach facilitates to model constraints for the graph transitions as part of the formal model, making an additional algorithm superfluous.

The research result is that the introduced approach resolves RQ2, based on the given assumptions discussed in section 4.9. The approach can be extended in future work to remove some of the assumptions.

RQ3: How to formalize a given system design concept and the requirements related to the safety and fault-tolerance concept of that system to be able to apply the formal analysis to this type of system?

In sections 4.5.1 and 4.6.5, we introduced formalized constraints that represent informal requirements about the considered safety and fault-tolerance concept shown in section 2.5.2, including redundancy and failover mechanisms. We have shown certain informal requirements and their formal representations in form of formal constraints over the introduced system model. The research result is that the introduced formal model allows to represent all considered requirements and even more, as we have not shown

all implemented requirements. In section 4.8, we introduced formalized optimization objectives for the synthesized deployments and channel selections within the considered failure scenarios.

5.2 Limitations of the presented approach

Subsequently, we discuss limitations of the approach that we introduced in chapter 4. Some of these limitations can be hurdled by extensions of our formal model, constraints and objectives as future work. Other limitations are more fundamental due to the assumed properties of the system under analysis and its assumed safety and redundancy concept, which hampers a transfer to certain other types of systems.

Behavior: We do not model the behavior of software components and the resulting behavior of functional features of the system. Instead, we focus on an analysis on structural level. The structural level allows to analyze which components can be executed in which scenarios, enabling to analyze which functional features can be kept available in these scenarios. However, this does not enable an analysis about the quality of service, in which a functional feature can be provided.

Quality of Communicated Data: We do not model different quality levels of communicated data. Such quality attributes of data might be used in connection with quality thresholds at subscription ports to specify degradation triggers of component behaviors. We also do not model different operating modes of software components, which may be used to represent different (potentially degraded) behaviors. However, in case a system under analysis contains software components with different operating modes related to different levels of quality of input data, such an extension of our model is a potential future work.

Black Box Software Components: We consider software components as black boxes, supporting no white box view into software components to identify causalities between subscription ports and publication ports of a software component. This restricts the introduced analysis to consider deactivations of entire components, as soon as one mandatory subscribed data item becomes unavailable. With an extension towards an internal causality model between subscription ports and publication ports of one component, meaning to model which subscription ports internally influence which publication ports, more fine grained deactivation and degradation strategies could be modeled and analyzed. Such an extension is future work. A concept for modeling and analyzing white-box causality relations of software components is presented for instance in [308].

Time: As discussed in section 4.9, our model does not incorporate a notion of time, for instance to model when in time a failure occurs. We also do not incorporate soft or hard real-time requirements for processor task scheduling, for network data transmissions, or for end-to-end timing constraints of data causality effects. Thus, we do not express temporal real-time requirements or *linear temporal logic* (LTL) formulas within the shown constraints. However, an extension of our approach towards the expression of temporal requirements by the introduced formal constraints would be possible as future work. For instance, it is already shown in [347] that temporal constraints for systems with a global discrete time base can be expressed as input for SMT solvers. We assume a logical execution time based system platform with homogeneous execution cycles, as well as a reliable real-time capable communication network, being able to distribute each communicated value to each receiver within a fixed amount of execution cycles. Due to the missing notion of time and time intervals, we do not calculate the *availability* of functional features as ratio between uptime and overall time, but instead we provide a boolean notion of *availability* in certain scenarios that may appear in some point in time.

Considered Failures: We focus on assumed internal failures of execution units and software components, as well as on structural reactions required due to isolations of these failed internal elements. The isolations are necessary to avoid failure propagation leading to an external failure at the system boundary. Although during runtime some failures may be *transient*, we only consider *permanent* failures. The error respectively failure detection and handling mechanism of the systems runtime environment may mask transient errors or failures, for instance by using last good known values for a limited amount of time. This is not considered in this thesis. We also do not consider healing of a system when a failed hardware or software component comes back to operation. However, this can be seen as a transition back to some predecessor scenario in the scenario graph (SG), being a potential future extension of the introduced analysis. However, in a complex system, there may exist more other failure modes which may be mandatory to be captured and analyzed, making our approach eventually insufficient or even inapplicable for these systems, if the model could not be extended to capture these additional failure modes.

Probabilistic Models: In most safety engineering approaches, like FTA and FMEA (see section 2.2), probabilistic models are used to describe the appearance of faults and the causality to errors and failures in order to analyze the system safety. However, in this thesis we do not support probabilistic modeling of the considered failures of system elements. Instead, we consider certain situations in which we assume certain failures to be occurred, without a notion of probability.

Scalability, Performance: The problem of finding optimal deployments is a NP-Hard problem [333]. Furthermore, as shown in section 4.6.10, the scenario graph can become very large, with an amount of scenario nodes progressively growing with the number of execution units and software components.

In this thesis, we set only a minor focus on reducing the complexity of the introduced formal model to allow efficient solving of the problem model. For instance, we merged the communication channels between ports of software components to communication channels between software components, in order to keep the communication matrices of the formal model smaller and thereby to improve the efficiency of finding solutions. We set no further focus on an efficient solving of the input problem model to improve scalability of our approach, as we used an out of the box SMT solver. Hence, in order to analyze large scale software and system architectures, alternative search and optimization technologies might be of interest, instead of the employed SMT solver. As future work, other more efficient problem solving and meta-heuristic optimization search strategies should be also employed and evaluated with respect to scalability for bigger models of the system under analysis. In section 2.7, we outlined alternative heuristic optimization strategies, providing non optimal but near optimal solutions in a more efficient amount of time. This is also relevant as other future work extensions of our model, as discussed above in this section or also below in section 6.3, would furthermore increase the size of the model.

Another future work to improve scalability would be to investigate if and how our analysis approach can be separated into multiple sub analyses, which can be solved independently and iteratively. This might potentially lead to a more efficient calculation of the analysis results, however potentially providing only approximative near optimal solutions, for instance for the required levels of degradation in the considered failure scenarios. Such separation of problems into iterative design space exploration has been for instance investigated in [153].

Assumed properties of system under analysis: In section 4.9.2, we discussed certain properties that we assume for the system under analysis. The presented analysis approach is tailored to system platforms that are similar to the platform that had been developed in the RACE project (see section 2.5.1), particularly in terms of scheduling, E/E architecture and communication topology, remote sensor/actuator access, and the presence of general purpose execution units.

This means, the presented formal model and analysis approach is not tailored for usual state-of-the-art distributed heterogeneous federated E/E architectures (see section 2.4.1), consisting of a communication topology with multiple heterogeneous bus systems, connected by a central gateway, as well as heterogeneous electronic control units, tailored for a special purpose.

Instead, our approach is tailored for future centralized automotive E/E architectures, comprising a centralized computing platform with a scalable amount of execution units. Such centralized E/E architectures are seen by many independent sources as future architectures to handle the challenges of autonomous driving, for instance in [133]. However, by using the *hwPlatform* property, we also support to synthesize deployments to heterogeneous execution units and also to peripheral execution units outside the central computing platform. In this way, our approach can also be extended to incorporate the analysis of effects of failures of sensors, which are attached to peripheral execution units respectively gateways.

Applicability to different types of systems: The applicability of the introduced formal model and analysis approach depends conceptually on the architectural style of the system under analysis, the used component model, model of execution, etc. The system under analysis has to match the above mentioned assumptions, therewith the model is able to adequately expresses all relevant system information. We assume for instance that the system under analysis applies an asynchronous publish/subscribe communication. Another limitation may be that the modeled set of functional features may be not precise enough to sufficiently analyze systems in which feature hierarchies (or function hierarchies) occupy a major role, like for instance discussed in [340]. In section 4.10.1, we discussed how we consider the relationship between feature sets and feature hierarchies. The same holds for hierarchical compositions of software components, what we also do not express in our formal model. The presented analysis approach is only applicable and valid for systems fulfilling the assumptions presented in section 4.9.2, and for systems whose software architecture uses a component model that is compatible with our formal model. Different component models are for instance discussed in [220].

5.3 Threats to Validity

Finally, the question arises whether the introduced formal model itself, and particularly the specified constraints, are valid with respect to correctly representing the system under analysis, and if the analysis results are correct as desired? Therefore, two questions arise when evaluating possible threats to validity to the introduced approach:

1. Is the introduced formal model valid? Does the approach really model that what is intended to be modeled? Is the model consistent and is it able to express the considered properties adequately and correctly?

2. Are the calculated solutions valid? If the solutions would be wrong, the analysis results would be wrong, as the solutions are the basis for our analysis.

Validity of the formal system model: The formal system model captures all that we intended to model as described in the research questions. We enriched the formal model by properties to express all system information needed to perform the introduced analysis, and we evaluated the models by the three shown examples, as well as by several smaller test models. If additional information would be of interest to analyze aspects in further examples, the model can be easily enriched by additional properties. The so far captured system information encompasses mixed-reliability by different levels of fail-operational requirements, mixed-criticality by different ASIL values, realization relationships between functional features and software components, communication channels between ports of software components, deployments of software components to hardware execution units incorporating a notion of redundancy,

and many more deployment relevant properties like WCETs and flash memory. The major motivation for the formal system model is to enable the expression of a set of formal constraints, describing valid deployments and valid reconfiguration scenarios and by this, allowing to analyze the system properties in which we are interested in, according to the discussed research questions.

Validity of the analysis solution: We do not develop an own algorithm to calculate solutions for our analysis results, but instead we employ an out of the box SMT solver to calculate solutions based on modeled formal constraints, which describe valid solutions. With respect to the correctness of the calculated solutions, we have to assume that the SMT solver itself is correct! The applied Z3 SMT solver developed at Microsoft Research is state of the art, applied at Microsoft in many areas (e. g., [138]), and therefore trusted to be well designed and well tested. Hence, finally the question arises if the modeled constraints describe valid solutions correctly?

Validity of the formal constraints: When considering the constraints for valid deployments, failovers and degradations, wrongly or incompletely modeled constraints may lead to invalid analysis results. Like in any other engineering approach, first of all the list of informal requirements, which lead to the creation of the formal constraints, may be incomplete, ambiguous, imprecise, inconsistent or simply wrong, what would result into formal constraints that do not reflect the correct complete and consistent requirements. The presented approach focuses on formalizing and formally analyzing a given set of informal requirements. The consistency between the informal requirements and their representation as formal constrains depends on the one hand on the quality of the informal requirements itself, and on the other hand on the quality and expressiveness of the underlying formal base model, which is used to express the formal constraints. We have shown three application example models in this thesis and have used a higher amount of heterogeneous small test models, with which we evaluated the correctness of the analysis results of our approach manually. During the evolution of the implementation of the model and the constraints, we used the set of test models and manually examine the analysis results to detect results that we rated as invalid. If such a situation appears, what was obviously most often the case when adding completely new constraints, or when changing the base model, we fixed the constraints to obtain valid analysis results for all test models. The same holds for the presented optimization objectives. We also applied a Python Unit-Test framework to automate these validity checks of the analysis results. However, of course the set of test models is finite, additional more complex test models may reveal further evaluation. But we can preclude major problems here due to the applied test and evaluation models.

Inconsistent requirements as well as inconsistent formalizations of requirements by formal constraints are partially detectable by obtaining an *unsatisfiable* returned by the SMT solver. But an unsatisfiable may also cause from an application software architecture that requires more resources than provided by the hardware architecture. With the help of the introduced relaxed soft constraints, some of the latter situations are detectable, because instead of an unsatisfiable, the solver returns a satisfied model in which some soft constraints are marked as unfulfilled. We leave it open to the user, which constraints he likes to relax in this way.

Finally, a formal verification of the refinement and consistency between informal requirements and their representation as formal constrains would be required. Refinement and consistency checks of models have been for instance tackled in [304]. Also techniques to analyze the quality of the informal requirements itself may be applied, like the approach to analyze informal natural language requirements introduced in [119].

CHAPTER 6 Conclusions and Future Work

This chapter summarizes the focus and contribution of this thesis and provides an outlook about future work that may be performed to improve and extend the approach introduced in this thesis.

Contents

6.1 Summary and Conclusions 159
6.2 Out of Scope 161
6.3 Future Work 161

6.1 Summary and Conclusions

In this thesis, we introduced a formal approach to analyze the fault-tolerance of embedded systems in the automotive domain. The main objective is to formally analyze if a software and system design is able to fulfill given heterogeneous *fail-operational* requirements in certain failure scenarios, and to determine necessary reactions to failures to ensure this. Fail-operational requirements increasingly arise when engineering autonomous X-by-Wire systems. Fully autonomous systems should consider as less human interaction as possible, as handover between autonomous and manual operation is difficult, particularly as humans are not always available. Think about a fully autonomous car, driving empty to pick you up at some place. X-by-Wire systems should not depend on mechanical fallbacks, as these fallbacks cause additional costs and need for physical space. Hence, we assume that human and mechanical fallbacks are no more involvable in fully autonomous X-by-Wire systems, which we aim to analyze.

We do not just analyze a given predefined software and system design, but support the construction of an appropriate design by automatically synthesizing certain design decisions in an open design space, leading to a successful analysis, if feasible. A feasible design denotes the fulfillment of all fail-operational requirements of functional features of the system under analysis, as well as the fulfillment of certain design and resource constraints.

Summarized, our contributions are:

- A formal system model including sets of functional features, software components and hardware execution units of a system, as well as additional attributes of these system elements and relationships between the elements, required to perform a failure effect analysis in different scenarios of failing execution units and failing software components.
- Encoding of formal constrains describing valid deployments of software components to hardware execution units, including constraints for valid redundancy mechanisms, valid failover and degradation scenarios, and other design aspects.
- Support of systems with mixed-criticality (by different ASIL) and mixed-reliability (by heterogeneous fail-operational requirements of functional features). The level of required fail-operationality denotes the amount of single system element failures that a functional feature shall survive.
- Synthesis of valid initial deployments of software to hardware, incorporating adequate minimal levels of redundancy needed to ensure the fulfillment of different fail-operational requirements of

functional features in different scenarios of failing system elements. We employ redundancy in form of a master/slave concept of software components, distinguishing hot and cold-standby slaves.

- Synthesis of valid reconfigurations of the deployments in failure scenarios to establish valid *failovers* between redundant software components, to ensure fail-operational requirements of those functional features that are realized by software components which become directly or indirectly affected by failures. For failover, we synthesize needed activations of redundantly deployed slave instances of software components.

- Synthesis of valid system *degradations* in case the system resources become insufficient in failure scenarios to provide the entire initial set of functional features. In this case, we determine a subset of software components that can be deactivated in order to release resources and thus to be able to keep the fail-operational features available with the remaining resources. During this, we aim in deactivating those components first which have the lowest requirements with respect to safety and reliability. Minimizing the loss of safety and reliability relevant components is our major optimization objective during the synthesis.

- Static structural analysis of the degree of necessary degradations in all considered failure scenarios. By construction, due to the formalized constraints, the synthesized architectural design properties ensure the fulfillment of all fail-operational requirements in all considered failure scenarios.

- Apart from degradations on system level (performed by deactivating functional features), we also support degradations on functional feature level. To model this, we distinguish between full fail-operational and degraded fail-operational functional features. The degraded fail-operationality is incorporated into our model by a mechanism to describe and apply degradations on feature level in form of substituting a full-fledged functional feature by a degraded version of that feature. We also model and apply degradations on software component level. This also enables to integrate *diversity* of components into the software architecture and switch between diverse components in case of failures.

- Apart from the synthesized deployments, we also synthesize communication channels between software components based on publish/subscribe definitions of ports of software components. If more channels are possible than required to serve all subscriptions with published data, we select a subset of communication channels that serves all subscriptions once. Hence, not all possible channels are used. We synthesize a selection of those channels which construct a communication structure between the software components that minimizes the required network traffic between execution units as a subordinate optimization objective. The synthesized channel selection as well as the deployments both influence the network traffic, as the deployment synthesis can prefer to establish local communication, not causing network traffic.

The validity of all synthesized design decisions is ensured by formal constraints over the introduced formal system model. For the deployments, degradations, and chosen communication channels, we optimize the synthesized design decisions by expressing optimization objectives.

To analyze effects of assumed failures of system elements onto the fulfillment of fail-operational requirements and onto the needed level of degradation to ensure this, we introduce a *scenario graph* into the formal model, in order to describe and analyze scenarios of assumed failures and subsequent isolations of execution units and/or software components. Valid transitions between scenarios are also expressed by formal constraints. For each failure scenario, we analyze the effect of lost software component instances to the availability of the functional features, and synthesize required failovers and explicit deactivations of software components to ensure that functional features with fail-operational requirements stay available. We apply a boolean availability statement for each functional feature in each scenario. The feature

availability is obtained by a tracing from software components to the functional features that are realized by a component. We perform the analysis on a structural level, without describing the behavior of software components or functional features. Summarized, our approach offers a static analysis of effects of failure scenarios on a structural architectural level. At design time of the system, this enables the identification of possible failure handling means.

The formal model and the constraints are expressed using arithmetic and logical expressions. We apply an out of the box SMT solver with optimization capabilities to obtain optimal solutions for the problem model, satisfying all constraints, if feasible. The hole scenario graph is incorporated into a single model, such that the applied SMT solver calculates a single solution model, which inherently ensures that the initial deployment allows valid degradation scenarios.

Conclusion: No existing work has been found that provides a similar combined formalization of architecture concepts and requirements for valid redundancy, failover and degradation mechanisms to enable fault-tolerance in the described manner.

We illustrate and evaluate the concepts and resulting scenarios of feature degradations and component failovers based on three self constructed examples. We assume some properties for the system under analysis. The applicability of our approach is limited to systems that hold the discussed assumptions.

By using the introduced approach for failure effect analysis, complex side-effects can be analyzed, like that software components might be required to become deactivated to be able to activate a passive backup of another redundant software component. This might result in the situation that a software component has to be deactivated, which was not deployed at all to a failed execution unit. But this deactivation might be necessary to ensure the fail-operational requirement of the functional feature that is realized by the other redundant software component. Hence, due to the needed deactivation, a functional feature might become unavailable which was not related at all to the failed execution unit. However, if we want to efficiently use resources and not integrate higher redundancy than necessary, such scenarios can appear, like shown in our examples. A manually performed analysis of such scenarios would be error prone, or even hardly achievable in case of large systems.

6.2 Out of Scope

What is not contained in our approach, is a description of functional behavior of software components or functional features. We also do not incorporate a notion of time and time intervals, or a notion of when in time failures appear or how probable failures are. Due to this, we calculate the availability of functional features in each failure scenario as a boolean availability metric for a certain point in time, not over a time interval like it is usually done as availability metric. We also do not consider concrete failure detection, isolation and failover mechanisms, which have to be applied at system runtime. In section 4.9, we discussed in more detail our assumptions and aspects that are out of scope.

6.3 Future Work

Several future work is possible related to the approach introduced in this thesis. We already discussed future work to diminish certain limitations of our approach in section 5.2. In this section we repeat this briefly, categorize the work, and extend it partially. We divide the future work into the following categories:

1. reducing the set of design aspects that are out of scope
2. extending the analysis approach by new capabilities
3. evaluation of alternative solving, optimization and design space exploration strategies

6.3.1 Expand the set of considered design aspects

As future work, several assumptions and out of scope design aspects may be mitigated, which we applied for the sake of simplicity in this thesis. This would let the analysis approach become more precise, and enables additional analysis features. We discussed the set of design aspects being out of scope in section 4.9.1 and 6.2. Extensions of the formal model would also mitigate limitations of our approach that we discussed in section 5.2. In that section, we also mentioned some future work to diminish these limitations. To expand the scope, our formal model needs to be extended to be able to express additional more detailed system aspects. For instance, this includes:

1. introduction of a notion of time and of probabilities of failures
2. introduction of a notion of behavior of functional features and software components, incorporating different qualities of behavior and exchanged data
3. introduction of different operation modes of functional features and software components, enriched by degradation relationships between different operation modes
4. modeling of the internal data-flow causality of software components, to support a more fine grained model about which missing input data affects which published output data. Based on this, instead of deactivating the whole component in case of a single missing mandatory subscribed input – as done in this thesis – the component might be switched into a different *operating mode* in which it can work with the reduced or degraded inputs, but produces only a subset of its outputs, potentially with less quality
5. support of multi-rate scheduling in addition to the so far supported single-rate scheduling

6.3.2 Possible Future Work Extensions of our Approach

The following extensions of our formal model may be done as future work to enlarge the capabilities of the introduced synthesis and analysis approach.

System Modes: One potential future work is the extension of our approach to support different *system modes*, each having different sets of required functional features and hence, also different sets of active software components. Our approach may be extended to determine optimized deployments with respect to efficient system mode transitions.

To enable this, the introduced formal system model may be extended with additional *System Modes*. For this, the introduced system model $\mathbb{S}$ could be extended by a *set of system modes* SM. Let $\text{SM} = \{m_1, ..., m_r\}$ denote a finite set of identifiers for different system modes. Each contained system mode $m_i \in \text{SM}$ might have specified a subset $F_{m_i} \subseteq F$ of functional features that should be active in this system mode. This means, functional features are only required in a subset of the system modes. This could be expressed as relationship by $\mu : \text{SM} \to \mathcal{P}(F)$ with $\mu(m) = \{f \in F \mid f \text{ should be active in system mode } m \in \text{SM}\}$ and $\mu^{-1} : F \to \mathcal{P}(\text{SM})$ with $\mu^{-1}(f) = \{m \in \text{SM} \mid f \in F \text{ should be active in system mode } m\}$.

By using the relation $\chi(f)$ about which ASWCs realize a given functional feature $f \in F$, we obtain also the ASWCs that are required in a certain system mode $m \in \text{SM}$, if $f \in \mu(m)$. We assume here that a functional feature is realized by the same ASWCs in all system modes. Hence, $\chi(f)$ does not change in different system modes. With the help of a system mode transition $\tau(m_o, m_n)$, constraints for the transition from an old system mode m_o to a new system mode m_n could be expressed. System modes in the automotive domain may be for instance manual driving (m_1) and autonomous driving (m_2). A precise elaboration of the system mode concept is out of scope of this thesis and left open as future work.

Analyze Plug-and-play scenarios at system runtime: As mentioned, the shown approach in this thesis is aimed to be applied at design time or during after-sale maintenance stops. As future work, the applicability of the approach at runtime by the system itself during plug-and-play scenarios might be considered and required changes identified. This would establish a basis to analyze fault-tolerance at runtime in order to make it "possible to upgrade a system during its lifetime and extend it with new features" [276]. However, to apply the analysis at runtime, more focus has to be spent on efficiency, see next future work section.

Our approach could also be extended to be used to prepare the deployment to be best suitable to cover future integrations of additional features, without having to change the existing deployment. This might be tackled by adding constraints which ensure that the utilization of execution time and memory space of the hardware execution units is preferably uniformly distributed, to obtain freedom for choices where to deploy new software components. However, if a redeployment (dynamic migration) of existing components is desired, this can also be done.

6.3.3 Evaluation of alternative solving and optimization strategies

Each of the discussed extensions may result in a higher complexity of the formal model, leading to decreased analysis efficiency. Hence, also heuristic problem solving approaches may be considered in future work, to obtain near optimal synthesis solutions, but exact analysis results, more efficient in less amount of time. Based on the list of satisfiability solving and optimization approaches presented in section 2.7, and also based on existing related approaches discussed in section 3.2, more efficient alternatives compared to the applied SMT solver should be evaluated as future work, while not forfeiting expressiveness in modeling the formal constraints.

CHAPTER A Appendix

A.1 Input Files of Examples

In sections 4.6.8, 4.6.9 and 4.7.5 we have shown three examples showing the applicability of the analysis approach introduced in this thesis.

Below, we show the problem definition files of these three examples as XML format, used as input for our analysis as shown in section 4.3 Fig. 4.13. Listing A.1 shows the XML file of example A (section 4.6.8), listing A.2 shows the XML file of example B (section 4.6.9), and listing A.3 shows the XML file of example C (section 4.7.5). The XML tags `<FEATURE>` describe the functional features $f \in F$, the tags `<SWC>` describe the application software components $s \in S$, and the tags `<HWC>` describe the execution units $e \in E$. The tag `<SYS>` contains the input model properties of the system configuration Φ. The property *maxNetworkTraffic* is not contained, because we made it dynamically configurable in the analysis tool, independently from the input model XML.

Listing A.1: XML file of Example A shown in section 4.6.8

```
<data>
    <SYS name="Example A">
        <faultRecoveryTime>50000</faultRecoveryTime>
    </SYS>

    <FEATURE name="NavigationSystem/Infotainment" id="0">
        <asil>0</asil>
        <failOp>0</failOp>
        <priority>1</priority>
        <realizedBySWC id="0"/>
    </FEATURE>

    <FEATURE name="Energymanagement" id="1">
        <asil>1</asil>
        <failOp>0</failOp>
        <priority>1</priority>
        <realizedBySWC id="1"/>
        <realizedBySWC id="2"/>
    </FEATURE>

    <FEATURE name="ADAS-A" id="2">
        <asil>3</asil>
        <failOp>0</failOp>
        <priority>1</priority>
        <realizedBySWC id="3"/>
        <realizedBySWC id="4"/>
    </FEATURE>

    <FEATURE name="ADAS-B" id="3">
        <asil>4</asil>
        <failOp>1</failOp>
        <priority>1</priority>
        <realizedBySWC id="4"/>
    </FEATURE>

```

```
36      <FEATURE name="ManualDriving" id="4">
37          <asil>4</asil>
38          <failOp>3</failOp>
39          <priority>1</priority>
40          <realizedBySWC id="5"/>
41          <realizedBySWC id="6"/>
42          <realizedBySWC id="7"/>
43      </FEATURE>
44
45      <!-- ASWCs -->
46
47      <SWC name="Infotainment" id="0">
48          <reqExecUnitHwPlatform>0</reqExecUnitHwPlatform>
49          <minimalFTT>99999</minimalFTT>
50          <domain>1</domain>
51          <wcet>2000</wcet>
52          <requiredROM>1</requiredROM>
53          <requiredRAM>1</requiredRAM>
54      </SWC>
55
56      <SWC name="RemainingRangeCalc" id="1">
57          <reqExecUnitHwPlatform>0</reqExecUnitHwPlatform>
58          <minimalFTT>99999</minimalFTT>
59          <domain>1</domain>
60          <wcet>700</wcet>
61          <requiredROM>1</requiredROM>
62          <requiredRAM>1</requiredRAM>
63      </SWC>
64
65      <SWC name="EnergyEfficiencyAssist" id="2">
66          <reqExecUnitHwPlatform>0</reqExecUnitHwPlatform>
67          <minimalFTT>99999</minimalFTT>
68          <domain>1</domain>
69          <wcet>300</wcet>
70          <requiredROM>1</requiredROM>
71          <requiredRAM>1</requiredRAM>
72      </SWC>
73
74      <SWC name="AdasSwc1" id="3">
75          <reqExecUnitHwPlatform>0</reqExecUnitHwPlatform>
76          <minimalFTT>99999</minimalFTT>
77          <domain>1</domain>
78          <wcet>1700</wcet>
79          <requiredROM>1</requiredROM>
80          <requiredRAM>1</requiredRAM>
81      </SWC>
82
83      <SWC name="AdasSwc2" id="4">
84          <reqExecUnitHwPlatform>0</reqExecUnitHwPlatform>
85          <minimalFTT>30000</minimalFTT>
86          <domain>1</domain>
87          <wcet>1000</wcet>
88          <requiredROM>1</requiredROM>
89          <requiredRAM>1</requiredRAM>
90      </SWC>
91
92      <SWC name="ManualAcceleration" id="5">
93          <reqExecUnitHwPlatform>0</reqExecUnitHwPlatform>
94          <minimalFTT>30000</minimalFTT>
95          <domain>1</domain>
96          <wcet>1000</wcet>
97          <requiredROM>1</requiredROM>
```

```
98            <requiredRAM>1</requiredRAM>
99        </SWC>
100
101       <SWC name="ManuelBraking" id="6">
102           <reqExecUnitHwPlatform>0</reqExecUnitHwPlatform>
103           <minimalFTT>30000</minimalFTT>
104           <domain>1</domain>
105           <wcet>1000</wcet>
106           <requiredROM>1</requiredROM>
107           <requiredRAM>1</requiredRAM>
108       </SWC>
109
110       <SWC name="ManualSteering" id="7">
111           <reqExecUnitHwPlatform>0</reqExecUnitHwPlatform>
112           <minimalFTT>30000</minimalFTT>
113           <domain>1</domain>
114           <wcet>500</wcet>
115           <requiredROM>1</requiredROM>
116           <requiredRAM>1</requiredRAM>
117       </SWC>
118
119       <!-- Execution Units -->
120
121       <HWC name="DCC 1">
122           <hwPlatform>0</hwPlatform>
123           <powerSupply>0</powerSupply>
124           <providedTimeBudget>4000</providedTimeBudget>
125           <providedFlashROM>64</providedFlashROM>
126       </HWC>
127
128       <HWC name="DCC 2">
129           <hwPlatform>0</hwPlatform>
130           <powerSupply>1</powerSupply>
131           <providedTimeBudget>4000</providedTimeBudget>
132           <providedFlashROM>64</providedFlashROM>
133       </HWC>
134
135       <HWC name="DCC 3">
136           <hwPlatform>0</hwPlatform>
137           <powerSupply>0</powerSupply>
138           <providedTimeBudget>4000</providedTimeBudget>
139           <providedFlashROM>64</providedFlashROM>
140       </HWC>
141
142       <HWC name="DCC 4">
143           <hwPlatform>0</hwPlatform>
144           <powerSupply>1</powerSupply>
145           <providedTimeBudget>4000</providedTimeBudget>
146           <providedFlashROM>64</providedFlashROM>
147       </HWC>
148   </data>
```

Listing A.2: XML file of Example B shown in section 4.6.9

```
<data>
    <SYS name="Example B">
        <faultRecoveryTime>50000</faultRecoveryTime>
    </SYS>


    <FEATURE name="Feature1" id="0">
        <asil>4</asil>
        <failOp>1</failOp>
        <priority>1</priority>
        <realizedBySWC id="0"/>
        <realizedBySWC id="1"/>
    </FEATURE>

    <FEATURE name="Feature2" id="1">
        <asil>2</asil>
        <failOp>0</failOp>
        <priority>1</priority>
        <realizedBySWC id="2"/>
    </FEATURE>

    <FEATURE name="Feature3" id="2">
        <asil>1</asil>
        <failOp>0</failOp>
        <priority>1</priority>
        <realizedBySWC id="3"/>
    </FEATURE>

    <FEATURE name="Feature4" id="3">
        <asil>0</asil>
        <failOp>0</failOp>
        <priority>1</priority>
        <realizedBySWC id="4"/>
    </FEATURE>

    <FEATURE name="Feature5" id="4">
        <asil>0</asil>
        <failOp>0</failOp>
        <priority>1</priority>
        <realizedBySWC id="5"/>
    </FEATURE>

    <FEATURE name="Feature6" id="5">
        <asil>0</asil>
        <failOp>0</failOp>
        <priority>1</priority>
        <realizedBySWC id="6"/>
    </FEATURE>

    <FEATURE name="Feature7" id="6">
        <asil>0</asil>
        <failOp>0</failOp>
        <priority>1</priority>
        <realizedBySWC id="7"/>
    </FEATURE>

    <!-- ASWCs -->

    <SWC name="s1" id="0">
        <reqExecUnitHwPlatform>0</reqExecUnitHwPlatform>
```

```
61          <minimalFTT>30000</minimalFTT> <!-- HOT-SLAVE -->
62          <domain>0</domain>
63          <wcet>1500</wcet>
64          <requiredROM>1</requiredROM>
65          <requiredRAM>1</requiredRAM>
66          <publication publPortID="0" name="s1.pub1" dataSize="1"/>
67          <publication publPortID="1" name="s1.pub2" dataSize="2"/>
68      </SWC>
69
70      <SWC name="s2" id="1">
71          <reqExecUnitHwPlatform>0</reqExecUnitHwPlatform>
72          <minimalFTT>60000</minimalFTT> <!-- COLD-SLAVE -->
73          <domain>1</domain>
74          <wcet>2500</wcet>
75          <requiredROM>1</requiredROM>
76          <requiredRAM>1</requiredRAM>
77          <subscription subPortID="0">
78              <!-- from s1.pub1 -->
79              <subscriptionMatch publComponentID="0" publPortID="0"/>
80          </subscription>
81      </SWC>
82
83      <SWC name="s3" id="2">
84          <reqExecUnitHwPlatform>0</reqExecUnitHwPlatform>
85          <minimalFTT>99999</minimalFTT>
86          <domain>2</domain>
87          <wcet>2000</wcet>
88          <requiredROM>1</requiredROM>
89          <requiredRAM>1</requiredRAM>
90          <publication publPortID="0" name="s3.pub1" dataSize="3"/>
91      </SWC>
92
93      <SWC name="s4" id="3">
94          <reqExecUnitHwPlatform>0</reqExecUnitHwPlatform>
95          <minimalFTT>99999</minimalFTT>
96          <domain>3</domain>
97          <wcet>2000</wcet>
98          <requiredROM>1</requiredROM>
99          <requiredRAM>1</requiredRAM>
100         <publication publPortID="0" name="s4.pub1" dataSize="4"/>
101     </SWC>
102
103     <SWC name="s5" id="4">
104         <reqExecUnitHwPlatform>0</reqExecUnitHwPlatform>
105         <minimalFTT>99999</minimalFTT>
106         <domain>4</domain>
107         <wcet>2000</wcet>
108         <requiredROM>1</requiredROM>
109         <requiredRAM>1</requiredRAM>
110         <subscription subPortID="0">
111             <!-- from s1.pub2 -->
112             <subscriptionMatch publComponentID="0" publPortID="1"/>
113         </subscription>
114         <subscription subPortID="1" isOptional="1">
115             <!-- from s4.pub1 (OPTIONAL) -->
116             <subscriptionMatch publComponentID="3" publPortID="0"/>
117         </subscription>
118         <subscription subPortID="2">
119             <!-- from s8.pub1 -->
120             <subscriptionMatch publComponentID="7" publPortID="0"/>
121         </subscription>
122     </SWC>
```

```
123
124    <SWC name="s6" id="5">
125        <reqExecUnitHwPlatform>0</reqExecUnitHwPlatform>
126        <minimalFTT>99999</minimalFTT>
127        <domain>5</domain>
128        <wcet>1000</wcet>
129        <requiredROM>1</requiredROM>
130        <requiredRAM>1</requiredRAM>
131        <publication publPortID="0" name="s6.pub1" dataSize="5"/>
132        <publication publPortID="1" name="s6.pub2" dataSize="6"/>
133        <subscription subPortID="0">
134            <!-- from s3.pub1 -->
135            <subscriptionMatch publComponentID="2" publPortID="0"/>
136        </subscription>
137    </SWC>
138
139    <SWC name="s7" id="6">
140        <reqExecUnitHwPlatform>0</reqExecUnitHwPlatform>
141        <minimalFTT>99999</minimalFTT>
142        <domain>6</domain>
143        <wcet>1000</wcet>
144        <requiredROM>1</requiredROM>
145        <requiredRAM>1</requiredRAM>
146        <publication publPortID="0" name="s7.pub1" dataSize="7"/>
147        <subscription subPortID="0" isOptional="1">
148            <subscriptionMatch publComponentID="2" publPortID="0"/>
149        </subscription>
150        <subscription subPortID="1">
151            <subscriptionMatch publComponentID="3" publPortID="0"/>
152        </subscription>
153    </SWC>
154
155    <SWC name="s8" id="7">
156        <reqExecUnitHwPlatform>0</reqExecUnitHwPlatform>
157        <minimalFTT>99999</minimalFTT>
158        <domain>7</domain>
159        <wcet>1000</wcet>
160        <requiredROM>1</requiredROM>
161        <requiredRAM>1</requiredRAM>
162        <publication publPortID="0" name="s8.pub1" dataSize="8"/>
163        <subscription subPortID="0" isOptional="1">
164            <subscriptionMatch publComponentID="5" publPortID="1"/>
165        </subscription>
166        <subscription subPortID="1">
167            <subscriptionMatch publComponentID="6" publPortID="0"/>
168        </subscription>
169    </SWC>
170
171    <!-- Execution Units -->
172
173    <HWC name="DCC 1">
174        <hwPlatform>0</hwPlatform>
175        <powerSupply>0</powerSupply>
176        <providedTimeBudget>4000</providedTimeBudget>
177        <providedFlashROM>64</providedFlashROM>
178    </HWC>
179
180    <HWC name="DCC 2">
181        <hwPlatform>0</hwPlatform>
182        <powerSupply>1</powerSupply>
183        <providedTimeBudget>4000</providedTimeBudget>
184        <providedFlashROM>64</providedFlashROM>
```

```
185         </HWC>
186
187         <HWC name="DCC 3">
188             <hwPlatform>0</hwPlatform>
189             <powerSupply>0</powerSupply>
190             <providedTimeBudget>4000</providedTimeBudget>
191             <providedFlashROM>64</providedFlashROM>
192         </HWC>
193
194         <HWC name="DCC 4">
195             <hwPlatform>0</hwPlatform>
196             <powerSupply>1</powerSupply>
197             <providedTimeBudget>4000</providedTimeBudget>
198             <providedFlashROM>64</providedFlashROM>
199         </HWC>
200 </data>
```

Listing A.3: XML file of Example C shown in section 4.7.5

```
<data>
    <SYS name="Example C">
        <faultRecoveryTime>50000</faultRecoveryTime>
    </SYS>

    <FEATURE name="Steer-By-Wire" id="0">
        <asil>4</asil>
        <failOp>0</failOp>
        <priority>1</priority>
        <degradedVersion>4</degradedVersion>
        <realizedBySWC id="0"/>
        <realizedBySWC id="1"/>
        <realizedBySWC id="2"/>
    </FEATURE>

    <FEATURE name="Parking Assistance (active)" id="1">
        <asil>3</asil>
        <failOp>0</failOp>
        <priority>1</priority>
        <degradedVersion>5</degradedVersion>
        <realizedBySWC id="2"/>
        <realizedBySWC id="3"/>
    </FEATURE>

    <FEATURE name="Drive-By-Wire" id="2">
        <asil>4</asil>
        <failOp>1</failOp>
        <priority>1</priority>
        <realizedBySWC id="4"/>
    </FEATURE>

    <FEATURE name="Infotainment" id="3">
        <asil>0</asil>
        <failOp>0</failOp>
        <priority>1</priority>
        <realizedBySWC id="5"/>
    </FEATURE>

    <!-- Degraded Versions of Features -->

    <FEATURE name="Steer-By-Wire (without assistance)" id="4">
        <asil>3</asil>
        <failOp>1</failOp>
        <priority>1</priority>
        <degradedVersion>6</degradedVersion>
        <realizedBySWC id="0"/>
        <realizedBySWC id="6"/>
    </FEATURE>

    <FEATURE name="Parking Assistance (passive)" id="5">
        <asil>3</asil>
        <failOp>1</failOp>
        <priority>1</priority>
        <realizedBySWC id="3"/>
        <realizedBySWC id="7"/>
    </FEATURE>

    <FEATURE name="Steer-By-Wire (limp home)" id="6">
        <asil>3</asil>
        <failOp>99</failOp>
```

```
61          <priority>1</priority>
62          <realizedBySWC id="8"/>
63      </FEATURE>
64
65      <!-- ASWCs -->
66
67      <SWC name="s1" id="0">
68          <reqExecUnitHwPlatform>0</reqExecUnitHwPlatform>
69          <minimalFTT>33333</minimalFTT>
70          <domain>1</domain>
71          <wcet>1500</wcet>
72          <requiredROM>10</requiredROM>
73          <requiredRAM>1</requiredRAM>
74          <publication name="Pub0.0(1)" publPortID="0" dataSize="1"/>
75      </SWC>
76
77      <SWC name="s2" id="1">
78          <reqExecUnitHwPlatform>0</reqExecUnitHwPlatform>
79          <minimalFTT>99999</minimalFTT>
80          <domain>1</domain>
81          <wcet>1000</wcet>
82          <requiredROM>10</requiredROM>
83          <requiredRAM>1</requiredRAM>
84          <degradedVersion>6</degradedVersion>
85          <subscription subPortID="0">
86              <subscriptionMatch publComponentID="0" publPortID="0"/>
87          </subscription>
88          <subscription subPortID="1">
89              <subscriptionMatch publComponentID="2" publPortID="0"/>
90          </subscription>
91      </SWC>
92
93      <SWC name="s3" id="2">
94          <reqExecUnitHwPlatform>0</reqExecUnitHwPlatform>
95          <minimalFTT>99999</minimalFTT>
96          <domain>1</domain>
97          <wcet>1000</wcet>
98          <requiredROM>10</requiredROM>
99          <requiredRAM>1</requiredRAM>
100         <degradedVersion>7</degradedVersion>
101         <publication name="Pub2.0(2)" publPortID="0" dataSize="2"/>
102         <publication name="Pub2.1(3)" publPortID="1" dataSize="4"/>
103     </SWC>
104
105     <SWC name="s4" id="3">
106         <reqExecUnitHwPlatform>0</reqExecUnitHwPlatform>
107         <minimalFTT>77777</minimalFTT>
108         <domain>1</domain>
109         <wcet>500</wcet>
110         <requiredROM>10</requiredROM>
111         <requiredRAM>1</requiredRAM>
112         <subscription subPortID="0" isOptional="1">
113             <subscriptionMatch publComponentID="2" publPortID="1"/>
114         </subscription>
115     </SWC>
116
117     <SWC name="s5" id="4">
118         <reqExecUnitHwPlatform>0</reqExecUnitHwPlatform>
119         <minimalFTT>77777</minimalFTT>
120         <domain>1</domain>
121         <wcet>1300</wcet>
122         <requiredROM>10</requiredROM>
```

```
123         <requiredRAM>1</requiredRAM>
124     </SWC>
125
126     <SWC name="s6" id="5">
127         <reqExecUnitHwPlatform>0</reqExecUnitHwPlatform>
128         <minimalFTT>99999</minimalFTT>
129         <domain>1</domain>
130         <wcet>500</wcet>
131         <requiredROM>17</requiredROM>
132         <requiredRAM>1</requiredRAM>
133     </SWC>
134
135     <!-- Degraded Versions of ASWCs -->
136
137     <SWC name="s2-degraded" id="6">
138         <reqExecUnitHwPlatform>0</reqExecUnitHwPlatform>
139         <minimalFTT>33333</minimalFTT>
140         <domain>1</domain>
141         <wcet>500</wcet>
142         <requiredROM>5</requiredROM>
143         <requiredRAM>1</requiredRAM>
144         <degradedVersion>8</degradedVersion>
145         <subscription subPortID="0">
146             <subscriptionMatch publComponentID="0" publPortID="0"/>
147         </subscription>
148     </SWC>
149
150     <SWC name="s3-degraded" id="7">
151         <reqExecUnitHwPlatform>0</reqExecUnitHwPlatform>
152         <minimalFTT>77777</minimalFTT>
153         <domain>1</domain>
154         <wcet>100</wcet>
155         <requiredROM>3</requiredROM>
156         <requiredRAM>1</requiredRAM>
157     </SWC>
158
159     <SWC name="s2-double-degraded" id="8">
160         <reqExecUnitHwPlatform>0</reqExecUnitHwPlatform>
161         <minimalFTT>33333</minimalFTT>
162         <domain>1</domain>
163         <wcet>100</wcet>
164         <requiredROM>2</requiredROM>
165         <requiredRAM>1</requiredRAM>
166     </SWC>
167
168     <!-- Execution Units -->
169
170     <HWC name="DCC 1">
171         <hwPlatform>0</hwPlatform>
172         <powerSupply>0</powerSupply>
173         <providedTimeBudget>4000</providedTimeBudget>
174         <providedFlashROM>64</providedFlashROM>
175     </HWC>
176
177     <HWC name="DCC 2">
178         <hwPlatform>0</hwPlatform>
179         <powerSupply>1</powerSupply>
180         <providedTimeBudget>4000</providedTimeBudget>
181         <providedFlashROM>64</providedFlashROM>
182     </HWC>
183 </data>
```

List of Figures

2.1 Dependability and its tree of attributes, threats and means [23] 9
2.2 Chains of threats, from Fig. 11 in [23] 11
2.3 Graceful Degradation classification into dependability means [300] 13
2.4 Illustrations for DMR, TMR and DDR from [154] 20
2.5 Redundancy characterization from [321] (Fig. 5.5) 21
2.6 Active I/Os of hot, warm and cold spares 22
2.7 Safety engineering lifecycle from [336] 23
2.8 AUTOSAR layered software architecture (Fig. 3.12 in `AUTOSAR_EXP_VFB.pdf` [22]) 26
2.9 AUTOSAR Virtual Function Bus and SW deployment tool (Fig. 2 in `AUTOSAR_TechnicalOverview.pdf` [20]) 27
2.10 Graceful degradation and other mechanisms for error handling at the software architectural level, from [176] (Table 5) 29
2.11 Example instance of the RACE hardware architecture [35] 31
2.12 RACE layered Architecture with RTE 32
2.13 Cyclic execution of RTE and ASWCs 32

3.1 Example Lattice from [246] 42
3.2 AC and OC self-organization loops 55

4.1 SPES Viewpoints 66
4.2 UML meta-model of the considered problem domain for the system under analysis . . . 68
4.3 Class diagram representation of the formal system model 71
4.4 Bijective mapping between sets of entities of real system and sets of identifiers of formal system model 72
4.5 ASWCs in a layered software architecture 73
4.6 Example ASWC with 3 subscription ports and 2 publication ports 74
4.7 Example of channels between ASWCs 75
4.8 Example of channels between ASWCs inclusive weights 76
4.9 Example of a mapping of ASWCs to ASWC-Clusters 77
4.10 Example of realization relationship between features and ASWCs 78
4.11 Example of a deployment of ASWCs to execution units 80
4.12 Example for the formal system model 81
4.13 Brief visualization of the tooling and the procedure how we use the introduced formal model 83
4.14 Class-diagram representation of the formal model, incl. properties as class attributes . . 85
4.15 Example of four ASWCs with some publications and mandatory subscriptions and possible communication channel candidates 90
4.16 Software component states 94
4.17 Example Scenario-Graph (SG) 100
4.18 Example of a system degradation after an execution unit isolation 102
4.19 Realization relationship $\chi(f_i)$ between functional features and ASWCs for the example of Table 4.2 108

4.20 Initial deployment for the example of Tab. 4.2 in scenario σ_0 110
4.21 Deployment after isolation of execution unit e_1 in scenario σ_1 112
4.22 Deployment matrices for scenarios σ_0 and σ_1 for the example shown in Tab. 4.2 112
4.23 Example set of ASWCs with published and subscribed data items d_i 113
4.24 Component architecture with communication channels 114
4.25 Initial deployment for the example of Fig. 4.23 in scenario σ_0 115
4.26 Deployment after isolation of execution unit e_1 in scenario σ_1 117
4.27 Deployment matrices for scenarios σ_0 and σ_1 for the example shown in Fig. 4.23 118
4.28 Example SG considering $n \leq 3$ isolations of 4 execution units, while considering the consecutive order of isolations 119
4.29 Example SG considering $n \leq 3$ isolations of 4 execution units, without considering the consecutive order of isolations 120
4.30 Example SG considering $n < 5$ isolations of 5 execution units, without considering the consecutive order of isolations 121
4.31 Example degradation relationships between full-fledged and degraded functional features 124
4.32 Example relation between normal and degraded ASWCs, realizing a full-fledged feature f_1 and the corresponding degraded feature f_1' 124
4.33 Another view on the example from Fig. 4.32, relation between normal and degraded ASWCs, realizing a full-fledged feature f_1 (green ellipse) and the corresponding degraded feature f_1' (orange ellipse) 125
4.34 Example for the definitions w.r.t. feature degradation 125
4.35 Example of a feature degradation over time 126
4.36 Class-diagram representation of the formal model, incl. degradations 127
4.37 Example for requirement R_{10} 128
4.38 Realization relationship $\chi(f_i)$ between functional features and ASWCs for the example of Table 4.12 130
4.39 Realization of full-fledged features f_1 and f_2, as well as corresponding degraded features f_1' and f_2' by partially shared ASWCs (f_1'' not shown) 130
4.40 Publication and subscription ports of ASWCs in the example 131
4.41 ASWC-Clusters and communication channels between ASWCs 131
4.42 An initial deployment solution for the example 132
4.43 Features with fail-operational requirements (example from Table 4.12) 134
4.44 Deployment in scenario σ_1 after isolation of execution unit e_1 135
4.45 Deployment in scenario σ_2 after isolation of execution unit e_2 136
4.46 Deployment in σ_3 after isolation of master of s_1 and failover to its hot-standby slave . . . 138
4.47 Deployment in scenario σ_4 after isolation of s_2 139
4.48 Deployment in scenario σ_5 after isolation of s_3 139
4.49 A hierarchical feature tree (left) and a set of functional features F (right) that is created based on the feature tree 144
4.50 RTE Broker that connects ports by channels 146

List of Tables

4.1 List of Formal Model Symbols . . . 82
4.2 Example set of functional features and the realizing ASWCS with some of the predefined properties . . . 108
4.3 Cluster mapping matrix $map(s_i, c_j)$ . . . 109
4.4 Calculation of $prioSumActiveASWCs(\sigma_0)$ . . . 110
4.5 Calculation of $prioSumActiveASWCs(\sigma_1)$ after isolation of e_1 . . . 111
4.6 Availability of functional features in the two shown scenarios for the example shown in Tab. 4.2 . . . 113
4.7 Matrix of mandatory channels $CM(s_i, s_k)$ for the example . . . 114
4.8 Matrix of optional channels $CO(s_i, s_k)$ for the example . . . 115
4.9 Calculation of $prioSumActiveASWCs(\sigma_0)$ . . . 116
4.10 Calculation of $prioSumActiveASWCs(\sigma_1)$ after isolation of e_1 . . . 117
4.11 Availability of functional features in the two shown scenarios for the example shown in Fig. 4.23 . . . 118
4.12 Example set of functional features and realizing software components . . . 129
4.14 Calculation of $prioSumAllASWCs$ . . . 133
4.15 Calculation of $prioSumActiveASWCs(\sigma_0)$ . . . 133
4.16 Calculation of $prioSumActiveASWCs(\sigma_1)$ . . . 135

Bibliography

[1] Asim Abdulkhaleq and Daniel Lammering. A Systematic Approach Based on STPA for Developing a Dependable Architecture for Fully Automated Driving. In *4th European STAMP Workshop 2016*, 2016.

[2] Asim Abdulkhaleq, Sebastian Vöst, Stefan Wagner, and John Thomas. An industrial case study on the evaluation of a safety engineering approach for software-intensive systems in the automotive domain. *Preprint Version*, 2016.

[3] Asim Abdulkhaleq and Stefan Wagner. Experiences with applying stpa to software-intensive systems in the automotive domain. In *2nd STAMP Workshop at MIT, Boston, USA*, 2013.

[4] Asim Abdulkhaleq and Stefan Wagner. Xstampp: an extensible stamp platform as tool support for safety engineering. In *4th STAMP Workshop at MIT, Boston, USA*, 2015.

[5] Daniel Adam, Joachim Fröschl, Uwe Baumgarten, Andreas Herkersdorf, and Hans-Georg Herzog. Cyber organic system-model - new approach for automotives system design. In *7th Int. Conf. on Adaptive and Self-Adaptive Systems and Applications (ADAPTIVE)*, 2015.

[6] Daniel Adam, Thomas Gehrsitz, and Uwe Baumgarten. Cyber organic systems network - a new network architecture for future vehicles. In *Internet of Things (WF-IoT), 2015 IEEE 2nd World Forum on*, pages 6–11. IEEE, 2015.

[7] P. Adelt, J. Donoth, J. Gausemeier, J. Geisler, S. Henkler, S. Kahl, B. Klöpper, A. Krupp, E. Münch, S. Oberthür, et al. Selbstoptimierende Systeme des Maschinenbaus - Definitionen, Anwendungen, Konzepte. *Heinz-Nixdorf-Institute, University of Paderborn*, 2008.

[8] Jose Aguilar and Erol Gelenbe. Task assignment and transaction clustering heuristics for distributed systems. *Information Sciences*, 97(1):199–219, 1997.

[9] Aldeida Aleti. Designing automotive embedded systems with adaptive genetic algorithms. *Automated Software Engineering*, 22(2):199–240, 2014.

[10] Husain Aljazzar, Manuel Fischer, Lars Grunske, Matthias Kuntz, Florian Leitner-Fischer, and Stefan Leue. Safety analysis of an airbag system using probabilistic FMEA and probabilistic counterexamples. In *Quantitative Evaluation of Systems, 2009. QEST'09. Sixth International Conference on the*, pages 299–308. IEEE, 2009.

[11] Christian Allmann, Manfred Broy, Mirko Conrad, Werner Damm, et al. *Automotive Roadmap Embedded Systems, Eingebettete Systeme in der Automobilindustrie, Roadmap 2015-2030*. GI, SafeTRANS, VDA, 2015.

[12] J. Andersson, R. De Lemos, S. Malek, and D. Weyns. Modeling dimensions of self-adaptive software systems. *Software Engineering for Self-Adaptive Systems*, pages 27–47, 2009.

[13] R. Anthony, A. Leonhardi, C. Ekelin, S. Burton, O. Redell, A. Weber, and V. Vollmer. A future dynamically reconfigurable automotive software system. *Moderne Elektronik im Kraftfahrzeug: Innovationen, Neuentwicklungen, Anwendungen, Praxisberichte*, 67:101, 2006.

[14] Richard Anthony, Achim Rettberg, Dejiu J. Chen, Isabell Jahnich, Gerrit de Boer, and Cecilia Ekelin. A dynamically reconfigurable automotive control system architecture. *Proceedings of the 17th International Federation of Automatic Control (IFAC) World Congress*, pages 9308–9313, 2008.

[15] Vincent Aravantinos, Sebastian Voss, Sabine Teufl, Florian Hölzl, and Bernhard Schätz. AutoFOCUS 3: Tooling concepts for seamless, model-based development of embedded systems. *Joint proceedings of ACES-MB*, page 19, 2015.

[16] Jean-Paul Arcangeli, Raja Boujbel, and Sébastien Leriche. Automatic deployment of distributed software systems: Definitions and state of the art. *Journal of Systems and Software (JSS)*, 2015.

[17] Michael Armbruster. *Eine fahrzeugübergreifende X-by-Wire Plattform zur Ausführung umfassender Fahr-und Assistenzfunktionen*. PhD thesis, Institute for Avionics Systems (Institut für Luftfahrtsysteme, ILS), University of Stuttgart, 2009.

[18] Michael Armbruster. Graceful degradation and fail-operational-support in the context of plug'n'play. *Safety@Siemens (Siemens internal)*, 2014.

[19] Michael Armbruster, Ludger Fiege, Gunter Freitag, Thomas Schmid, Gernot Spiegelberg, and Andreas Zirkler. Ethernet-Based and Function-Independent Vehicle Control-Platform: Motivation, Idea and Technical Concept Fulfilling Quantitative Safety-Requirements from ISO 26262. *Adv. Microsystems for Automotive Applications (AMAA)*, pages 91–107, 2012.

[20] AUTOSAR. AUTomotive Open System ARchitecture (v3.2 Rev1). `http://www.autosar.org/specifications/release-32`.

[21] AUTOSAR. AUTomotive Open System ARchitecture (v4.1 Rev2). `http://www.autosar.org/specifications/release-41`.

[22] AUTOSAR. AUTomotive Open System ARchitecture (v4.2.2). `http://www.autosar.org/specifications/release-42`, 2015.

[23] A. Avizienis, J.C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11–33, 2004.

[24] Thomas Bäck and Hans-Paul Schwefel. An overview of evolutionary algorithms for parameter optimization. *Evolutionary computation*, 1(1):1–23, 1993.

[25] Massimo Baleani, Alberto Ferrari, Leonardo Mangeruca, Alberto Sangiovanni-Vincentelli, Maurizio Peri, and Saverio Pezzini. Fault-tolerant platforms for automotive safety-critical applications. In *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, pages 170–177. ACM, 2003.

[26] Joseph A Bannister and Kishor S Trivedi. Task allocation in fault-tolerant distributed systems. *Acta Informatica*, 20(3):261–281, 1983.

[27] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard Version 2.0, 2010.

[28] Peter A Barrett and Neil A Speirs. Towards an integrated approach to fault tolerance in delta-4. *Distributed Systems Engineering*, 1(2):59, 1993.

[29] Sanjoy K Baruah, Alan Burns, and Robert I Davis. Response-time analysis for mixed criticality systems. In *IEEE 32nd Real-Time Systems Symposium (RTSS)*, pages 34–43. IEEE, 2011.

[30] Twan Basten, Martijn Hendriks, Nikola Trčka, Lou Somers, Marc Geilen, Yang Yang, Georgeta Igna, Sebastian de Smet, Marc Voorhoeve, Wil van der Aalst, et al. Model-driven design-space exploration for software-intensive embedded systems. In *Model-Based Design of Adaptive Embedded Systems*, pages 189–244. Springer, 2013.

[31] Twan Basten, Emiel Van Benthum, Marc Geilen, Martijn Hendriks, Fred Houben, Georgeta Igna, Frans Reckers, Sebastian De Smet, Lou Somers, Egbert Teeselink, et al. Model-driven design-space exploration for embedded systems: the octopus toolset. In *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, pages 90–105. Springer, 2010.

[32] Benoit Baudry and Martin Monperrus. The multiple facets of software diversity: Recent developments in year 2000 and beyond. *ACM Computing Surveys (CSUR)*, 48(1):16, 2015.

[33] Basil Becker, Holger Giese, Stefan Neumann, Martin Schenck, and Arian Treffer. Model-based extension of AUTOSAR for architectural online reconfiguration. In *Models in Software Engineering, ACES-MB Workshop*, pages 83–97. Springer LNCS 6002, 2009.

[34] Jan Becker and Michael Helmle. Architecture and system safety requirements for automated driving. In *Road Vehicle Automation 2*, pages 37–48. Springer, 2015.

[35] Klaus Becker, Michael Armbruster, Bernhard Schätz, and Christian Buckl. Deployment calculation and analysis for a fail-operational automotive platform. In *1st Workshop on Engineering Dependable Systems of Systems (EDSoS)*. arXiv:1404.7763, 2014.

[36] Klaus Becker, Jelena Frtunikj, Meik Felser, Ludger Fiege, Christian Buckl, Stefan Rothbauer, Licong Zhang, and Cornel Klein. RACE RTE: A Runtime Environment for Robust Fault-Tolerant Vehicle Functions. In *3rd Workshop on Critical Automotive Applications : Robustness & Safety (CARS)*. HAL archives ouvertes, 2015.

[37] Klaus Becker and Bernhard Schätz. Deployment calculation and analysis for a fault-tolerant system platform. In *11th Dagstuhl-Workshop on Model-Based Development of Embedded Systems (MBEES)*, 2015.

[38] Klaus Becker, Bernhard Schätz, Michael Armbruster, and Christian Buckl. A formal model for constraint-based deployment calculation and analysis for fault-tolerant systems. In *12th Int. Conference on Software Engineering and Formal Methods (SEFM)*, volume 8702, pages 205–219. Springer Lecture Notes in Computer Science (LNCS), 2014.

[39] Klaus Becker and Sebastian Voss. Towards Dynamic Deployment Calculation for Extensible Systems using SMT-Solvers. In *First Open EIT ICT Labs Workshop on Cyber-Physical Systems Engineering (EIT CPSE)*, Trento, Italy, May 2013.

[40] Klaus Becker and Sebastian Voss. Analyzing graceful degradation for mixed critical fault-tolerant real-time systems. In *18th International Symposium on Real-Time Distributed Computing (ISORC)*, pages 110–118. IEEE, 2015.

[41] Klaus Becker and Sebastian Voss. A formal model and analysis of feature degradation in fault-tolerant systems. In *4th Int. Workshop on Formal Techniques for Safety-Critical Systems (FTSCS)*, volume 596, pages 139–154. Springer Communications in Computer and Information Science (CCIS), 2015.

[42] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010.

[43] David Benavides, Antonio Ruiz-Cortes, Pablo Trinidad, and Sergio Segura. A survey on the automated analyses of feature models. In *Jornadas de Ingenieria del Software y Bases de Datos JISBD 2006*, 2006.

[44] Andrew L Benjamin and Jaynarayan H Lala. Advanced fault tolerant computing for future manned space missions. In *Digital Avionics Systems Conference, 1997. 16th DASC., AIAA/IEEE*, volume 2, pages 8–5. IEEE, 1997.

[45] Benz & Co. Fahrzeug mit Gasmotorantrieb. *Kaiserliches Patentamt, Patentschrift DRP Nr. 37435*, 1886.

[46] Peter Bergmiller. Design and safety analysis of a drive-by-wire vehicle. In *Automotive Systems Engineering*, pages 147–202. Springer, 2013.

[47] Bernhard, M., Buckl, C., Döricht, V., Fehling, M., Fiege, L., von Grolman, H., Ivandic, N., Janelle, C., Klein, C., Kuhn, K.-J., Patzlaff, C., Riedl, B., Schätz, B., Stanek, C. *The Software Car: Information and Communication Technology (ICT) as an Engine for the Electromobility of the Future*. fortiss GmbH, 2011. Summary of results of the "eCar ICT System Architecture for Electromobility" research project sponsored by the German Federal Ministry of Economics and Technology (BMWi).

[48] D. Bertrand, A.M. Déplanche, S. Faucou, and O.H. Roux. A study of the aadl mode change protocol. In *Engineering of Complex Computer Systems, 2008. ICECCS 2008. 13th IEEE International Conference on*, pages 288–293. IEEE, 2008.

[49] Nikolaj Bjørner. Satisfiability: From quality to quantities (invited talk). In *20th Int. Conf on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 9450. Springer Lecture Notes in Computer Science (LNCS), 2015.

[50] Nikolaj Bjørner and Leonardo de Moura. Applications of smt solvers to program verification. *Rough notes for SSFT 2014*, 2014.

[51] Nikolaj Bjørner and Anh-Dung Phan. νZ - Maximal Satisfaction with Z3. In *6th Int. Symposium on Symbolic Computation in Software Science (SCSS)*, volume 30, pages 1–9. EasyChair Proceedings in Computing (EPiC), 2014.

[52] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. νZ - An Optimizing SMT Solver. *21st Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2015.

[53] Mogens Blanke, Marcel Staroswiecki, and N Eva Wu. Concepts and methods in fault-tolerant control. In *Proceedings of the American Control Conference*, volume 4, pages 2606–2620. IEEE, 2001.

[54] Hagen Böhmert and Pierre Blüher. Nachweis der funktionalen Integrität für automatisierte Fahrfunktionen. *HANSER automotive, Continental AG - Division Automotive*, 10:24 – 26, 2016.

[55] Bas Boone, Filip De Turck, and Bart Dhoedt. Automated deployment of distributed software components with fault tolerance guarantees. In *6th International Conference on Software Engineering Research, Management and Applications (SERA)*, pages 21–27. IEEE, 2008.

[56] J Börcsök, M Schwarz, E Ugljesa, P Holub, and A Hayek. High-availability controller concept for steering systems: the degradable safety controller. In *in WSEAS conference, Tenerife, Spain*, 2011.

[57] Josef Börcsök. Electronic safety systems, hardware concepts, models, and calculations, huthig gmbh & co. *KG Heidelberg, Germany*, 2004.

[58] Etienne Borde, Grégory Haïk, and Laurent Pautet. Mode-based reconfiguration of critical software component architectures. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1160–1165. IEEE, 2009.

[59] Marco Bozzano, Alessandro Cimatti, Joost-Pieter Katoen, Viet Yen Nguyen, Thomas Noll, and Marco Roveri. Safety, dependability and performance analysis of extended aadl models. *The Computer Journal*, 2010.

[60] Marco Bozzano and Adolfo Villafiorita. *Design and safety assessment of critical systems*. CRC press, 2010.

[61] J.S. Bradbury, J.R. Cordy, J. Dingel, and M. Wermelinger. A survey of self-management in dynamic software architecture specifications. In *Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, pages 28–33. ACM, 2004.

[62] Sally C Brailsford, Chris N Potts, and Barbara M Smith. Constraint satisfaction problems: Algorithms and applications. *European Journal of Operational Research*, 119(3):557–581, 1999.

[63] Jürgen Branke, Moez Mnif, Christian Müller-Schloer, Holger Prothmann, Urban Richter, Fabian Rochner, and Hartmut Schmeck. Organic computing–addressing complexity by controlled self-organization. In *Leveraging Applications of Formal Methods, Verification and Validation, 2006. ISoLA 2006. Second International Symposium on*, pages 185–191. IEEE, 2006.

[64] Francisco V Brasileiro, Paul Devadoss Ezhilchelvan, Santosh K Shrivastava, Neil Speirs, Sha Tao, et al. Implementing fail-silent nodes for distributed systems. *Computers, IEEE Transactions on*, 45(11):1226–1238, 1996.

[65] U. Brinkschulte, M. Pacher, and A. Renteln. An artificial hormone system for self-organizing real-time task allocation in organic middleware. *Organic Computing*, pages 261–283, 2008.

[66] M Broy, E Geisberger, MV Cengarle, P Keil, J Niehaus, C Thiel, and HJ Thönnißen-Fries. Cyber-physical systems: Innovationsmotor für mobilität, gesundheit, energie und produktion. acatech bezieht position, vol. 8, 2011.

[67] Manfred Broy. Challenges in automotive software engineering. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, pages 33–42. ACM, 2006.

[68] Manfred Broy, Sascha Kirstan, Helmut Krcmar, Bernhard Schätz, and Jens Zimmermann. What is the benefit of a model-based design of embedded software systems in the car industry? *Software Design and Development: Concepts, Methodologies, Tools, and Applications: Concepts, Methodologies, Tools, and Applications*, page 310, 2013.

[69] Manfred Broy and Ketil Stølen. Specification and development of interactive systems. *Monographs in Computer Science, Springer-Verlag*, 2001.

[70] C. Buckl, A. Camek, G. Kainz, C. Simon, L. Mercep, H. Stahle, and A. Knoll. The software car: Building ICT architectures for future electric vehicles. In *Electric Vehicle Conference (IEVC), 2012 IEEE International*, pages 1–8. IEEE, 2012.

[71] Christian Buckl, Michael Geisinger, Dhiraj Gulati, Franz Ruiz-Bertol, and Alois Knoll. Chromosome - a run-time environment for plug & play-capable embedded real-time systems. In *Workshop on Adaptive and Reconfigurable Embedded Systems (APRES)*, 2014.

[72] Martin Buechel, Jelena Frtunikj, Klaus Becker, Stephan Sommer, Christian Buckl, Michael Armbruster, Cornel Klein, Andre Marek, Andreas Zirkler, and Alois Knoll. An automated electric vehicle prototype showing new trends in automotive architectures. In *IEEE 18th International Conference on Intelligent Transportation Systems (ITSC)*, 2015.

[73] Alan Burns and Robert Davis. Mixed Criticality Systems – A Review. Technical report, Department of Computer Science, University of York, 7th edition, 2016.

[74] Giorgio C Buttazzo. *Hard Real-Time Computing Systems: predictable scheduling algorithms and applications*. Springer Science & Business Media, 3rd edition, 2011.

[75] Antonio Cansado, Carlos Canal, Gwen Salaun, and Javier Cubo. A formal framework for structural reconfiguration of components under behavioural adaptation. *Electronic Notes in Theoretical Computer Science*, 263:95–110, 2010.

[76] Miguel Castro and Barbara Liskov. Proactive recovery in a byzantine-fault-tolerant system. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation-Volume 4*, pages 19–19. USENIX Association, 2000.

[77] C. Cetina, V. Pelechano, P. Trinidad, and A. Cortes. An architectural discussion on dspl. In *Proceedings of the 12th International Software Product Line Conference (SPLC 2008)*, pages 59–68, 2008.

[78] Samarjit Chakraborty, Martin Lukasiewycz, Christian Buckl, Suhaib Fahmy, Naehyuck Chang, Sangyoung Park, Younghyun Kim, Patrick Leteinturier, and Hans Adlkofer. Embedded systems and software challenges in electric vehicles. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 424–429. EDA Consortium, 2012.

[79] D. Chen, R. Anthony, M. Persson, S. Scholle, V. Friesen, G. de Boer, A. Rettberg, and C. Ekelin. An architectural approach to autonomics and self-management of automotive embedded electronic systems. *Proceedings of the 4th European Congress ERTS (EMBEDDED REAL TIME SOFTWARE)*, 2008.

[80] Liming Chen and Algirdas Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Digest of Papers FTCS-8: Eighth Annual International Conference on Fault Tolerant Computing*, pages 3–9, 1978.

[81] B. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, et al. Software engineering for self-adaptive systems: A research roadmap. *Software Engineering for Self-Adaptive Systems. Dagstuhl Seminar Proceedings.*, pages 1–26, 2009.

[82] Chih-Hong Cheng, Saddek Bensalem, Harald Ruess, Natarajan Shankar, and Ashish Tiwari. EFSMT: A Logical Framework for the Design of Cyber-Physical Systems. In *Cyber-Physical System Architectures and Design Methodologies (CPSArch)*, 2014.

[83] Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. Computing small unsatisfiable cores in satisfiability modulo theories. *Journal of Artificial Intelligence Research*, pages 701–728, 2011.

[84] David W Coit. Cold-standby redundancy optimization for nonrepairable systems. *Iie Transactions*, 33(6):471–478, 2001.

[85] David W Coit. Maximization of system reliability with a choice of redundancy strategies. *IIE transactions*, 35(6):535–543, 2003.

[86] Louise K Comfort, Arjen Boin, and Chris C Demchak. *Designing resilience: Preparing for extreme events*. University of Pittsburgh Pre, 2010.

[87] Thomas Crick. *Superoptimisation: provably optimal code generation using answer set programming*. PhD thesis, University of Bath, 2009.

[88] Ivica Crnkovic and Lars Grunske. Evaluating dependability attributes of component-based specifications. In *Companion to the proceedings of the 29th International Conference on Software Engineering*, pages 157–158. IEEE Computer Society, 2007.

[89] Vanderlei da Costa Bueno and Iran Martins do Carmo. Active redundancy allocation for a k-out-of-n: F system of dependent components. *European Journal of Operational Research*, 176(2):1041–1051, 2007.

[90] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[91] John C Day, Michel D Ingham, Richard M Murray, Leonard J Reder, and Brian C Williams. Engineering resilient space systems. *INSIGHT*, 18(1):23–25, 2015.

[92] R. De Lemos, H. Giese, H.A. Muller, M. Shaw, J. Andersson, L. Baresi, B. Becker, N. Bencomo, Y. Brun, B. Cukic, et al. Software engineering for self-adaptive systems: A second research roadmap. *Software Engineering for Self-Adaptive Systems. Dagstuhl Seminar Proceedings.*, pages 1–16, 2011.

[93] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.

[94] Tom De Wolf and Tom Holvoet. Emergence and self-organisation: a statement of similarities and differences. *Engineering Self-Organising Systems*, 3464:1–15, 2004.

[95] A. Dearle. Software deployment, past, present and future. In *Future of Software Engineering (FOSE)*, pages 269–284. IEEE Computer Society, 2007.

[96] Michael S Deutsch and Ronald R Willis. *Software quality engineering: a total technical and management approach*. Prentice Hall Englewood Cliffs, NJ, 1988.

[97] G. Di Marzo Serugendo, M.P. Gleizes, and A. Karageorgos. Self-organisation and emergence in mas: An overview. *Informatica*, 30(1):45–54, 2006.

[98] Giovanna Di Marzo Serugendo, John Fitzgerald, Alexander Romanovsky, and Nicolas Guelfi. A metadata-based architectural model for dynamically resilient systems. In *Proceedings of the 2007 ACM symposium on Applied computing*, pages 566–572. ACM, 2007.

[99] Michael Dinkel. *A Novel IT-Architecture for Self-Management in Distributed Embedded Systems*. PhD thesis, Technical University of Munich, 2008.

[100] Michael Dinkel and Uwe Baumgarten. Self-configuration of vehicle systems-algorithms and simulation. In *Proceedings of the 4th International Workshop on Intelligent Transportation (WIT)*, pages 85–91, 2007.

[101] T. Dinkelaker, R. Mitschke, K. Fetzer, and M. Mezini. A dynamic software product line approach using aspect models at runtime. In *Proceedings of the 9th IEEE International Conference on Computer and Information Technology*, pages 11–18, 2010.

[102] Tobias Distler, Ivan Popov, Wolfgang Schröder-Preikschat, Hans P Reiser, and Rüdiger Kapitza. Spare: Replicas on hold. In *NDSS*, 2011.

[103] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. Ant system: optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 26(1):29–41, 1996.

[104] Bruce Powel Douglass. *Real-time design patterns: robust scalable architecture for real-time systems*, volume 1. Addison-Wesley Professional, 2003.

[105] Joanne Bechta Dugan, Salvatore J Bavuso, Mark Boyd, et al. Dynamic fault-tree models for fault-tolerant computer systems. *Reliability, IEEE Transactions on*, 41(3):363–377, 1992.

[106] Bruno Dutertre. Solving Exists/Forall Problems With Yices. In *Workshop on Satisfiability Modulo Theories*, 2015.

[107] Bruno Dutertre and Leonardo De Moura. The Yices SMT solver. Technical report, SRI International, 2006.

[108] Klaus Echtle. *Fehlertoleranzverfahren*. Springer, 1990.

[109] Klaus Echtle and Thorsten Kimmeskamp. Fault-tolerant and fail-safe control systems-using remote redundancy. In *Architecture of Computing Systems (ARCS), 2009 22nd International Conference on*, pages 1–6. VDE, 2009.

[110] Johannes Eder. Usable Design Space Exploration in AutoFOCUS3. In *Workshop on Open Source Software for Model-Driven Engineering (OSS4MDE), in conjunction with MODELS conference*, 2016.

[111] Cecilia Ekelin. *An optimization framework for scheduling of embedded real-time systems*. Chalmers University of Technology,, 2004.

[112] Robert J Ellison, David A Fisher, Richard C Linger, Howard F Lipson, and Thomas Longstaff. Survivable network systems: An emerging discipline. Technical report, DTIC Document, 1997.

[113] Paul Emberson. *Searching for flexible solutions to task allocation problems*. PhD thesis, University of York, 2009.

[114] Paul Emberson and Iain Bate. Extending a task allocation algorithm for graceful degradation of real-time distributed embedded systems. In *Real-Time Systems Symposium (RTSS)*, pages 270–279. IEEE, 2008.

[115] Stephan Esch and Bardo Lang. Elektronik-und Vernetzungsarchitektur mit gesteigerter Leistungsfähigkeit. *ATZextra*, 2(13):194–199, 2008.

[116] Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131, 2003.

[117] Stefan Faulstich, Berthold Hahn, and Peter J Tavner. Wind turbine downtime and its importance for offshore deployment. *Wind Energy*, 14(3):327–337, 2011.

[118] Marcus Fehling and Michael Armbruster. Major characteristics of a new ict system architecture for electric vehicles: Technology leadership brief. Technical report, SAE Technical Paper, 2012.

[119] Henning Femmer, Daniel Méndez Fernández, Stefan Wagner, and Sebastian Eder. Rapid quality assurance with requirements smells. *Journal of Systems and Software*, 123:190—213, 2017.

[120] L. Feng, D.J. Chen, and M. Torngren. Self configuration of dependent tasks for dynamically reconfigurable automotive embedded systems. In *Decision and Control, 2008. CDC 2008. 47th IEEE Conference on*, pages 3737–3742. IEEE, 2008.

[121] Carlos M Fonseca and Peter J Fleming. An overview of evolutionary algorithms in multiobjective optimization. *Evolutionary computation*, 3(1):1–16, 1995.

[122] R Freitag, M Moser, M Hartl, J Koepernik, and L Eckstein. Anforderungen an das Sicherheitskonzept von Lenksystemen mit Steer-by-Wire Funktionalität / Safety concept requirements of steering systems with steer-by-wire functionality. *VDI-Berichte*, 2001.

[123] S. Fritsch, A. Senart, D.C. Schmidt, and S. Clarke. Time-bounded adaptation for automotive system software. In *Proceedings of the 30th international conference on Software engineering*, pages 571–580. ACM, 2008.

[124] Joachim Froehlich and Reiner Schmid. Architecture for a Hard-Real-Time System Enabling Non-intrusive Tests. In *2014 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 24–24, 2014.

[125] Joachim Fröhlich, Jelena Frtunikj, Stefan Rothbauer, and Christoph Stückjürgen. Testing safety properties of cyber-physical systems with non-intrusive fault injection – an industrial case study. In *Workshop on Dependable Embedded and Cyber-physical Systems and Systems-of-Systems (DECSoS) at International Conference on Computer Safety, Reliability, and Security (SAFECOMP)*, volume 9923, pages 105–117. Springer, 2016.

[126] Jelena Frtunikj. Safety framework and platform for functions of future automotive e/e systems. *Automotive and Engine Technology*, pages 1–13, 2016.

[127] Jelena Frtunikj, Michael Armbruster, and Alois Knoll. Run-time adaptive error and state management for open automotive systems. *International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2014.

[128] Jelena Frtunikj, Vladimir Rupanov, Michael Armbruster, and Alois Knoll. Adaptive error and sensor management for autonomous vehicles: Model-based approach and run-time system. In *Model-Based Safety and Assessment*, pages 166–180. Springer, 2014.

[129] Jelena Frtunikj, Vladimir Rupanov, Alexander Camek, Christian Buckl, and Alois Knoll. A safety aware run-time environment for adaptive automotive control systems. In *Embedded Real-Time Software and Systems (ERTS2)*, 2014.

[130] Christopher P Fuhrman, Sailesh Chutani, and Henri J Nussbaumer. A fault-tolerant implementation using multiple-task triple modular redundancy. In *Factory Communication Systems, 1995. WFCS'95, Proceedings., 1995 IEEE International Workshop on*, pages 75–80. IEEE, 1995.

[131] Simon Fürst. Challenges in the design of automotive software. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 256–258. European Design and Automation Association, 2010.

[132] Simon Fürst. Autosar the next generation – the adaptive platform. In *3rd Workshop on Critical Automotive Applications : Robustness & Safety (CARS)*. Keynote, 2015.

[133] Roland Galbas and Andreas Lock. Trends in E/E-Architectures. In *21st SafeTRANS Industrial Day*. Robert Bosch GmbH, 2016.

[134] Jürgen Gausemeier, Franz Josef Rammig, Wilhelm Schäfer, and Walter Sextro. *Dependability of self-optimizing mechatronic systems*. Springer, 2014.

[135] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *ICLP/SLP*, volume 88, pages 1070–1080, 1988.

[136] Michael Glaß, Martin Lukasiewycz, Christian Haubelt, and Jürgen Teich. Incorporating graceful degradation into embedded system design. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, pages 320–323. IEEE, 2009.

[137] Fred Glover. Future paths for integer programming and links to artificial intelligence. *Computers & operations research*, 13(5):533–549, 1986.

[138] Patrice Godefroid. 500 Machine-Years of Software Model Checking and SMT Solving. 12th Int. Conference on Software Engineering and Formal Methods (SEFM), 2014.

[139] A. Gomes, T. Batista, A. Joolia, and G. Coulson. Architecting dynamic reconfiguration in dependable systems. In *Architecting dependable systems IV*, pages 237–261. Springer LNCS 4615, 2007.

[140] O. González, H. Shrikumar, J.A. Stankovic, and K. Ramamritham. Adaptive fault tolerance and graceful degradation under dynamic hard real-time scheduling. In *Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE*, pages 79–89. IEEE, 1997.

[141] Sebastian Graf, Michael Glaß, Jürgen Teich, and Christoph Lauer. Design space exploration for automotive e/e architecture component platforms. In *Digital System Design (DSD), 2014 17th Euromicro Conference on*, pages 651–654. IEEE, 2014.

[142] Sebastian Graf, Sebastian Reinart, Michael Glaß, Jürgen Teich, and Daniel Platte. Robust design of e/e architecture component platforms. In *Design Automation Conference (DAC)*, 2015.

[143] Rudolf Grave and Alexander Much. Auf alles vorbereitet sein - Architekturen und Degradationsmechanismen für verlässliches Verhalten im Fehlerfall. `http://www.elektroniknet.de/automotive/assistenzsysteme/artikel/121101`, 2015.

[144] Jim Gray. Why do computers stop and what can be done about it? In *Symposium on reliability in distributed software and database systems*, pages 3–12. Los Angeles, CA, USA, 1985.

[145] Jim Gray and Daniel P Siewiorek. High-availability computer systems. *Computer*, 24(9):39–48, 1991.

[146] Lars Grunske. Identifying good architectural design alternatives with multi-objective optimization strategies. In *Proceedings of the 28th international conference on Software engineering*, pages 849–852. ACM, 2006.

[147] Lars Grunske, Robert Colvin, and Kirsten Winter. Probabilistic model-checking support for fmea. In *Quantitative Evaluation of Systems, 2007. QEST 2007. Fourth International Conference on the*, pages 119–128. IEEE, 2007.

[148] Xiaozhe Gu, Arvind Easwaran, Kieu My Phan, and Insik Shin. Resource efficient isolation mechanisms in mixed-criticality scheduling. In *27th Euromicro Conference on Real-Time Systems (ECRTS)*, 2015.

[149] Pragya Kirti Gupta, Klaus Becker, Markus Duchon, and Bernhard Schätz. Formalizing performance degradation strategies as an enabler for self-healing smart energy systems. In *11th Dagstuhl-Workshop on Model-Based Development of Embedded Systems (MBEES)*, 2015.

[150] Pragya Kirti Gupta and Bernhard Schätz. Constraint-based graceful degradation in smart grids. In *2nd Int. Workshop on Software Engineering for Smart Cyber-Physical Systems (SEsCPS)*, 2016.

[151] Julian Hall and Qi Huangfu. A high performance dual revised simplex solver. In *Parallel Processing and Applied Mathematics*, pages 143–151. Springer, 2011.

[152] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. Dynamic software product lines. *Computer*, 41(4):93–95, 2008.

[153] Arne Hamann. *Iterative design space exploration and robustness optimization for embedded systems*. PhD thesis, TU Braunschweig, 2008.

[154] Robert Hammett. Design by extrapolation: an evaluation of fault-tolerant avionics. In *Digital Avionics Systems, 2001. DASC. 20th Conference*, volume 1, pages 1C5–1. IEEE, 2001.

[155] B. Hardung, T. Kölzow, and A. Krüger. Reuse of software in distributed embedded automotive systems. In *Proceedings of the 4th ACM international conference on Embedded software*, pages 203–210. ACM, 2004.

[156] Bernd Hardung. *Optimisation of the allocation of functions in vehicle networks*. PhD thesis, University of Erlangen-Nuremberg, 2006.

[157] Rick H Hay, Clarence S Smith, Robert D Girts, and Larry J Yount. Fail-operational fault tolerant flight critical computer architecture and monitoring method, 1996. US Patent 5,550,736.

[158] Ábel Hegedüs, Ákos Horváth, and Dániel Varró. A model-driven framework for guided design space exploration. *Automated Software Engineering*, 22(3):399–436, 2015.

[159] Harald Heinecke, Anton Schedl, Josef Berwanger, Martin Peller, Volker Nieten, R Belschner, Bernd Hedenetz, Peter Lohrmann, and Claas Bracklo. FlexRay – ein Kommunikationssystem für das Automobil der Zukunft. *Elektronik Automotive*, 9, 2002.

[160] Abdelsalam A Helal, Abdelsalam A Heddaya, and Bharat K Bhargava. *Replication techniques in distributed systems*, volume 4. Springer, 1996.

[161] T. Henzinger, B. Horowitz, and C. Kirsch. Giotto: A time-triggered language for embedded programming. In *Embedded Software*, pages 166–184. Springer, 2001.

[162] Maurice P Herlihy and Jeannette M Wing. Specifying graceful degradation in distributed systems. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 167–177. ACM, 1987.

[163] Maurice P Herlihy and Jeannette M Wing. Specifying graceful degradation. *Parallel and Distributed Systems, IEEE Transactions on*, 2(1):93–104, 1991.

[164] Fernando Herrera, Héctor Posadas, Pablo Penil, Eugenio Villar, Francisco Ferrero, Raúl Valencia, and Gianluca Palermo. The complex methodology for uml/marte modeling and design space exploration of embedded systems. *Journal of Systems Architecture (JSA)*, 60(1):55–78, 2014.

[165] Abbas Heydarnoori and Walter Binder. A graph-based approach for deploying component-based applications into channel-based distributed environments. *Journal of Software*, 6(8):1381–1394, 2011.

[166] Scott A Hissam, Gabriel A Moreno, Judith A Stafford, and Kurt C Wallnau. Packaging predictable assembly. In *International Working Conference on Component Deployment*, pages 108–124. Springer, 2002.

[167] Pierre-Emmanuel Hladik, Hadrien Cambazard, Anne-Marie Déplanche, and Narendra Jussien. Solving a real-time allocation problem with constraint programming. *Journal of Systems and Software (JSS)*, 81(1):132–149, 2008.

[168] Lok Man Ho and Daniel Ossmann. Fault detection and isolation of vehicle dynamics sensors and actuators for an overactuated x-by-wire vehicle. In *53rd IEEE Conference on Decision and Control*, 2014.

[169] Kai Höfig, Marc Zeller, and Konstantin Schorp. Automated failure propagation using inner port dependency traces. In *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)*, pages 123–128. ACM, 2015.

[170] JH Holland and DE Goldberg. Genetic algorithms in search, optimization and machine learning, 1989.

[171] Albert L Hopkins Jr, T Smith III, and Jaynarayan H Lala. FTMP—A highly reliable fault-tolerant multiprocessor for aircraft. *Proceedings of the IEEE*, 66(10):1221–1239, 1978.

[172] James J Horning, Hugh C Lauer, Peter M Melliar-Smith, and Brian Randell. *A program structure for error detection and recovery*. Springer, 1974.

[173] IEEE 802.1 Time-Sensitive Networking Task Group. Time-Sensitive Networks (TSN). `http://www.ieee802.org/1/pages/tsn.html`.

[174] International Organization for Standardization (ISO). ISO 26262 - Road vehicles - Functional safety. Technical report, Technical Committee 22 (ISO/TC 22), Geneva, Switzerland, 2011.

[175] International Organization for Standardization (ISO). ISO 26262-1 - Road vehicles - Functional safety, Part 1: Glossary. Technical report, Technical Committee 22 (ISO/TC 22), 2011.

[176] International Organization for Standardization (ISO). ISO 26262-6 - Road vehicles - Functional safety, Part 6: Product development at the software level. Technical report, Technical Committee 22 (ISO/TC 22), 2011.

[177] International Organization for Standardization (ISO). ISO 26262-9 - Road vehicles - Functional safety, Part 9: ASIL oriented and safety-oriented analyses. Technical report, Technical Committee 22 (ISO/TC 22), 2011.

[178] International Organization for Standardization (ISO) JTC 1/SC 7. ISO/IEC 25010 - Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models. `http://www.iso.org/iso/catalogue_detail.htm?csnumber=35733`, 2011.

[179] International Organization for Standardization (ISO), TC 22/SC 31. ISO 11898 - Road vehicles – Controller area network (CAN). `http://www.iso.org/iso/catalogue_detail.htm?csnumber=63648`, 2015.

[180] International Organization for Standardization (ISO), TC 22/SC 31. ISO/DIS 17987 - Road vehicles – Local Interconnect Network (LIN). `http://www.iso.org/iso/catalogue_detail.htm?csnumber=61222`, 2015.

[181] Rolf Isermann. Model-based fault-detection and diagnosis – status and applications. *Annual Reviews in control*, 29(1):71–85, 2005.

[182] Rolf Isermann, Ralf Schwarz, and Stefan Stolzl. Fault-tolerant drive-by-wire systems. *Control Systems, IEEE*, 22(5):64–81, 2002.

[183] Viacheslav Izosimov, Paul Pop, Petru Eles, and Zebo Peng. Design optimization of time-and cost-constrained fault-tolerant distributed embedded systems. In *Design, Automation and Test in Europe (DATE)*, pages 864–869. IEEE Computer Society, 2005.

[184] V. Januzaj, S. Kugele, F. Biechele, and R. Mauersberger. A Configuration Approach for IMA Systems. In *10th International Conference on Software Engineering and Formal Methods (SEFM)*, 2012.

[185] Weijia Jia and Wanlei Zhou. Reliability and replication techniques. *Distributed Network Systems: From Concepts to Implementations*, pages 213–254, 2005.

[186] Maximilian Junker. *Specification and Analysis of Availability for Software-Intensive Systems*. PhD thesis, Technical University of Munich, 2016.

[187] Muhammad Kafil and Ishfaq Ahmad. Optimal task assignment in heterogeneous distributed computing systems. *Concurrency, IEEE*, 6(3):42–50, 1998.

[188] Bernhard Kaiser and Catharina Gramlich. State-event-fault-trees–a safety analysis model for software controlled systems. In *Computer Safety, Reliability, and Security*, pages 195–209. Springer, 2004.

[189] Oliver S. Kaiser, Heinz Eickenbusch, Vera Grimm, and Axel Zweck. Zukunft des Autos. *VDI Technologiezentrum - Zukünftige Technologien Consulting*, 75, 2008.

[190] Robert Kaiser and Stephan Wagner. Evolution of the PikeOS microkernel. In *Proceedings of the 1st International Workshop on Microkernels for Embedded Systems*, pages 50–57, 2007.

[191] Peter Kali and Stein W Wallace. *Stochastic programming*. Springer, 1994.

[192] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, DTIC Document, 1990.

[193] Zádor Dániel Kelemen, Jos JM Trienekens, Rob J Kusters, and Katalin Balla. A process based unification of process-oriented software quality approaches. In *ICGSE*, pages 285–288, 2009.

[194] James Kennedy and Russell Eberhart. Particle swarm optimization. In *International Conference on Neural Networks*, volume 4, pages 1942–1948. IEEE, 1995.

[195] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[196] Junsung Kim, Gaurav Bhatia, Ragunathan Raj Rajkumar, and Markus Jochim. Safer: System-level architecture for failure evasion in real-time applications. In *Real-Time Systems Symposium (RTSS), 2012 IEEE 33rd*, pages 227–236. IEEE, 2012.

[197] Junsung Kim, Ragunathan Raj Rajkumar, and Markus Jochim. Towards dependable autonomous driving vehicles: a system-level approach. *ACM SIGBED Review*, 10(1):29–32, 2013.

[198] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[199] Scott Kirkpatrick. Optimization by simulated annealing: Quantitative studies. *Journal of statistical physics*, 34(5-6):975–986, 1984.

[200] Christoph M Kirsch and Ana Sokolova. The logical execution time paradigm. In *Advances in Real-Time Systems*, pages 103–120. Springer, 2012.

[201] C. Klein, R. Schmid, C. Leuxner, W. Sitou, and B. Spanfelner. A survey of context adaptation in autonomic computing. In *Autonomic and Autonomous Systems, 2008. ICAS 2008. Fourth International Conference on*, pages 106–111. IEEE, 2008.

[202] John C Knight and Kevin J Sullivan. On the definition of survivability. *University of Virginia, Department of Computer Science, Technical Report CS-TR-33-00*, 2000.

[203] Andre Kohn, Michael Kasmeyer, Rolf Schneider, Andre Roger, Claus Stellwag, and Andreas Herkersdorf. Fail-operational in safety-related automotive multi-core systems. In *Industrial Embedded Systems (SIES), 2015 10th IEEE International Symposium on*, pages 1–4. IEEE, 2015.

[204] Abdullah Konak, David W Coit, and Alice E Smith. Multi-objective optimization using genetic algorithms: A tutorial. *Reliability Engineering & System Safety*, 91(9):992–1007, 2006.

[205] Antoaneta Kondeva, Daniel Ratiu, Bernhard Schätz, and Sebastian Voss. Seamless model-based development of embedded systems with af3 phoenix. In *ECBS*, page 212, 2013.

[206] Philip Koopman. Elements of the self-healing system problem space. In *Workshop on Software Architectures for Dependable Systems (WADS) at ICSE*, 2003.

[207] Philip Koopman. *Better Embedded System Software*. Carnegie Mellon University, 2010.

[208] Hermann Kopetz. Fault containment and error detection in the time-triggered architecture. In *Autonomous Decentralized Systems, 2003. ISADS 2003. The Sixth International Symposium on*, pages 139–146. IEEE, 2003.

[209] Hermann Kopetz. *Real-time systems: design principles for distributed embedded applications*. Springer Science & Business Media, 2011.

[210] Stefan Kugele and Gheorghe Pucea. Model-based optimization of automotive E/E-architectures. In *6th Int. Workshop on Constraints in Software Testing, Verification, and Analysis (CSTVA)*, pages 18–29. Association for Computing Machinery (ACM), 2014.

[211] Stefan Kugele, Gheorghe Pucea, Ramona Popa, Laurent Dieudonne, and Horst Eckardt. On the deployment problem of embedded systems. *13th ACM-IEEE International Conference on Formal Methods and Models for System Design*, 2015.

[212] Per A Kullstam. Availability, mtbf and mttr for repairable m out of n system. *Reliability, IEEE Transactions on*, 30(4):393–394, 1981.

[213] Ranjan Kumar, Kazuhiro Izui, Yoshimura Masataka, and Shinji Nishiwaki. Multilevel redundancy allocation optimization using hierarchical genetic algorithm. *Reliability, IEEE Transactions on*, 57(4):650–661, 2008.

[214] Vipin Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI magazine*, 13(1):32, 1992.

[215] Simon Kunzli. *Efficient design space exploration for embedded systems*. PhD thesis, Swiss Federal Institute of Technology Zurich, Switzerland, 2006.

[216] Daniel Lammering, Norbert Balbierer, and Asim Abdulkhaleq. Automatisiertes Fahren: Keimzelle neuer Architekturkonzepte. *HANSER automotive, Continental AG - Division Automotive*, 11-12:34 – 37, 2016.

[217] Jean-Claude Laprie. Dependable computing: Concepts, limits, challenges. In *FTCS-25, the 25th IEEE International Symposium on Fault-Tolerant Computing-Special Issue*, pages 42–54, 1995.

[218] Waldemar F Larsen. Fault tree analysis. Technical report, DTIC Document, 1974.

[219] Tudor A Lascu, Jacopo Mauro, and Gianluigi Zavattaro. Automatic deployment of component-based applications. *Science of Computer Programming (SCP), Special Issue on Formal Aspects of Component Software (FACS 2013)*, 113:261–284, 2015.

[220] Kung-Kiu Lau and Zheng Wang. Software component models. *IEEE Transactions on software engineering*, 33(10):709–724, 2007.

[221] Michael Lauer, Matthieu Amy, William Excoffon, Matthieu Roy, and Miruna Stoicescu. Towards Adaptive Fault Tolerance: From a Component-Based Approach to ROS. In *3rd Workshop on Critical Automotive Applications - Robustness & Safety (CARS)*, 2015.

[222] J. Lee and D. Muthig. Feature-oriented variability management in product line engineering. *Communications of the ACM*, 49(12):55–59, 2006.

[223] Nancy Leveson, Mirna Daouk, Nicolas Dulac, and Karen Marais. A systems theoretic approach to safety engineering. *Dept. of Aeronautics and Astronautics, Massachusetts Inst. of Technology, Cambridge*, 2003.

[224] Gregory Levitin, Liudong Xing, Barry Johnson, and Yuanshun Dai. Mission reliability, cost and time for cold standby computing systems with periodic backup. *The IEEE Transactions on Computers (TC)*, 64(4):1043 – 1057, 2015.

[225] Thomas Liebetrau, Ursula Kelling, Tobias Otter, and Magnus Hell. White Paper - Energy Saving in Automotive E/E Architectures. Technical report, Infineon Technologies AG, 2012.

[226] N. Loughran, P. Sánchez, A. Garcia, and L. Fuentes. Language support for managing variability in architectural models. In *Software Composition*, pages 36–51. Springer, 2008.

[227] Martin Lukasiewycz. *Modeling, analysis, and optimization of automotive networks*. PhD thesis, Ingolstadt Institute of Friedrich-Alexander-University Erlangen-Nuremberg and AUDI AG (INI.FAU), 2010.

[228] Martin Lukasiewycz, Michael Glaß, Christian Haubelt, and Jürgen Teich. Efficient symbolic multi-objective design space exploration. In *13th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 691–696. IEEE Computer Society Press, 2008.

[229] Martin Lukasiewycz, Florian Sagstetter, and Sebastian Steinhorst. Efficient design space exploration of embedded platforms. In *Proceedings of the 52nd Design Automation Conference (DAC)*, 2015.

[230] Martin Lukasiewycz, Sebastian Steinhorst, Sidharta Andalam, Florian Sagstetter, Peter Waszecki, Wanli Chang, Matthias Kauer, Philipp Mundhenk, Shreejith Shanker, Suhaib A Fahmy, et al. System architecture and software design for electric vehicles. In *Proceedings of the 50th Annual Design Automation Conference*, page 95. ACM, 2013.

[231] Frank D Macías-Escrivá, Rodolfo Haber, Raul del Toro, and Vicente Hernandez. Self-adaptive systems: A survey of current approaches, research challenges and applications. *Expert Systems with Applications*, 40(18):7267–7279, 2013.

[232] Jan Madsen, Thomas K Stidsen, Peter Kjærulf, and Shankar Mahadevan. Multi-objective design space exploration of embedded system platforms. In *From Model-Driven Design to Resource Management for Distributed Embedded Systems*, pages 185–194. Springer, 2006.

[233] Edward Mahinda and Brian Whitworth. Evaluating flexibility and reliability in emergency response information systems. In *Proceedings of ISCRAM2004*, pages 93–98, 2004.

[234] S. Malek, N. Medvidovic, and M. Mikic-Rakic. An extensible framework for improving a distributed software system's deployment architecture. *Software Engineering, IEEE Transactions on*, 38(1):73–100, 2012.

[235] Panagiotis Manolios, Daron Vroon, and Gayatri Subramanian. Automating component-based system assembly. In *Proceedings of the 2007 international symposium on Software testing and analysis (ISSTA)*, pages 61–72. ACM, 2007.

[236] N. Medvidovic and R.N. Taylor. A classification and comparison framework for software architecture description languages. *Software Engineering, IEEE Transactions on*, 26(1):70–93, 2000.

[237] I. Meedeniya, A. Aleti, and L. Grunske. Architecture-driven reliability optimization with uncertain model parameters. *Journal of Systems and Software*, 2012.

[238] I. Meedeniya, B. Buhnova, A. Aleti, and L. Grunske. Reliability-driven deployment optimization for embedded systems. *Journal of Systems and Software*, 84(5):835–846, 2011.

[239] Marija Mikic-Rakic, Sam Malek, Nels Beckman, and Nenad Medvidovic. A tailorable environment for assessing the quality of deployment architectures in highly distributed settings. In *Component Deployment*, pages 1–17. Springer, 2004.

[240] Martin Mitzlaff, Rüdiger Kapitza, and Wolfgang Schröder-Preikschat. Enabling mode changes in a distributed automotive system. In *Proceedings of the 1st Workshop on Critical Automotive applications: Robustness & Safety*, pages 75–78. ACM, 2010.

[241] MOST Cooperation. MOST Media Oriented Systems Transport — multimedia and control networking technology. `http://www.mostcooperation.com`, 1998.

[242] Adrien Mouaffo, Davide Taibi, and Kavyashree Jamboti. Controlled experiments comparing fault-tree-based safety analysis techniques. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, page 46. ACM, 2014.

[243] Henry Muccini, Mohammad Sharaf, and Danny Weyns. Self-adaptation for cyber-physical systems: A systematic literature review. In *11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, 2016.

[244] G. Mühl, M. Werner, M.A. Jaeger, K. Herrmann, and H. Parzyjegla. On the definitions of self-managing and self-organizing systems. In *Communication in Distributed Systems (KiVS), 2007 ITG-GI Conference*, pages 1–11. VDE, 2007.

[245] Nils Müllner. *Unmasking fault tolerance: Quantifying deterministic recovery dynamics in probabilistic environments*. PhD thesis, University of Oldenburg, 2014.

[246] William Nace. *Automatic Graceful Degradation for Distributed Embedded Systems*. PhD thesis, Carnegie Mellon University (CMU), 2002.

[247] William Nace and Philip Koopman. A product family approach to graceful degradation. In *International IFIP WG 10.3 / WG 10.4 / WG 10.5 Workshop on Distributed and Parallel Embedded Systems (DIPES)*, 2000.

[248] William Nace and Philip Koopman. A graceful degradation framework for distributed embedded systems. In *Workshop on Reliability in Embedded Systems*, 2001.

[249] NASA Goddard Space Flight Center (GSFC). Flight assurance procedure: Performing a failure mode and effects analysis. 431-REF-000370, Number P-302-720, 1996.

[250] Nicolas Navet and Françoise Simonot-Lion. In-vehicle communication networks-a historical perspective and review. *Industrial Communication Technology Handbook, Second Edition*, 2013.

[251] Victor P. Nelson. Fault-tolerant computing: Fundamental concepts. *Computer*, 23(7):19–25, 1990.

[252] Moritz Neukirchner, Steffen Stein, Harald Schrom, and Rolf Ernst. Self-configuration in hard realtime systems. Demonstration at International Conference on Autonomic Computing (ICAC), 2011.

[253] Koji Nonobe and Toshihide Ibaraki. A tabu search approach to the constraint satisfaction problem as a general problem solver. *European Journal of Operational Research*, 106(2):599–623, 1998.

[254] Jaewon Oh, Hyokyung Bahn, Chisu Wu, and Kern Koh. Pareto-based soft real-time task scheduling in multiprocessor systems. In *Software Engineering Conference, 2000. APSEC 2000. Proceedings. Seventh Asia-Pacific*, pages 24–28. IEEE, 2000.

[255] OMG. Modeling and analysis of real-time embedded systems, uml profile for marte v1.1. `http://www.omg.org/spec/MARTE`.

[256] P. Oreizy, N. Medvidovic, and R.N. Taylor. Architecture-based runtime software evolution. In *Proceedings of the 20th international conference on Software engineering*, pages 177–186. IEEE Computer Society, 1998.

[257] P. Oreizy, N. Medvidovic, and R.N. Taylor. Runtime software adaptation: framework, approaches, and styles. In *Companion of the 30th international conference on Software engineering*, pages 899–910. ACM, 2008.

[258] Gianluca Palermo, Cristina Silvano, and Vittorio Zaccaria. Multi-objective design space exploration of embedded systems. *Journal of Embedded Computing*, 1(3):305–316, 2005.

[259] G. Pardo-Castellote, B. Farabaugh, and R. Warren. An introduction to dds and data-centric communications, 2005.

[260] Paul Parkinson and Larry Kinnan. Safety-critical software development for integrated modular avionics. *Embedded System Engineering*, 11(7):40–41, 2003.

[261] Michael Paulitsch. Mixed-criticality systems - a journey "embedded" in time and space. In *27th Euromicro Conference on Real-Time Systems (ECRTS), Keynote*, 2015.

[262] Dar-Tzen Peng, Kang G Shin, and Tarek F Abdelzaher. Assignment and scheduling communicating periodic tasks in distributed real-time systems. *Software Engineering, IEEE Transactions on*, 23(12):745–758, 1997.

[263] Dulcineia Penha, Gereon Weiß, and Alexander Stante. Pattern-based approach for designing fail-operational safety-critical embedded systems. In *Int. Conference on Embedded and Ubiquitous Computing (EUC)*, pages 52–59. IEEE, 2015.

[264] Haapanen Pentti and Helminen Atte. Failure mode and effects analysis of software-based automation systems. *VTT Industrial Systems, STUK-YTO-TR 190*, 2002.

[265] Soila Pertet and Priya Narasimhan. Proactive recovery in distributed corba applications. In *Dependable Systems and Networks, 2004 International Conference on*, pages 357–366. IEEE, 2004.

[266] Adrian Petcu and Boi Faltings. A scalable method for multiagent constraint optimization. Technical report, École Polytechnique Fédérale de Lausanne (EPFL), 2005.

[267] Claudio Pinello, Luca P Carloni, and Alberto L Sangiovanni-Vincentelli. Fault-tolerant deployment of embedded software for cost-sensitive real-time feedback-control applications. In *Proceedings of the conference on Design, automation and test in Europe (DATE) - Volume 2*, page 21164. IEEE Computer Society, 2004.

[268] Claudio Pinello, Luca P Carloni, and Alberto L Sangiovanni-Vincentelli. Fault-tolerant distributed deployment of embedded control software. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(5):906–919, 2008.

[269] Klaus Pohl, Manfred Broy, Heinrich Daembkes, and Harald Hönninger. *Advanced Model-Based Engineering of Embedded Systems: Extensions of the SPES 2020 Methodology*. Springer, 2016.

[270] Uwe Pohlmann and Marcus Hüwe. Model-driven allocation engineering. *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015.

[271] Uwe Pohlmann, Matthias Meyer, Andreas Dann, and Christopher Brink. Viewpoints and views in hardware platform modeling for safe deployment. In *Proceedings of the 2nd Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling (VAO)*, page 23. ACM, 2014.

[272] Florian Pölzlbauer. *Kommunikationszentrierte Softwareverteilung zum Entwurf und Erweitern von verteilten echtzeitfähigen eingebetteten Systemen*. PhD thesis, TU Graz, 2014.

[273] Florian Pölzlbauer, Iain Bate, and Eugen Brenner. Software deployment for distributed embedded real-time systems of automotive applications. In *Embedded and Real Time System Development: A Software Engineering Perspective*, pages 305–328. Springer, 2014.

[274] Paul Pop, Viacheslav Izosimov, Petru Eles, and Zebo Peng. Design optimization of time-and cost-constrained fault-tolerant embedded systems with checkpointing and replication. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 17(3):389–402, 2009.

[275] Dhiraj K Pradhan and Nitin H Vaidya. Roll-forward checkpointing scheme: A novel fault-tolerant architecture. *Computers, IEEE Transactions on*, 43(10):1163–1174, 1994.

[276] Alexander Pretschner, Manfred Broy, Ingolf H. Kruger, and Thomas Stauner. Software engineering for automotive systems: A roadmap. In *Future of Software Engineering (FOSE)*, FOSE '07, pages 55–71, Washington, DC, USA, 2007. IEEE Computer Society.

[277] Paul J Prisaznuk. Arinc 653 role in integrated modular avionics (ima). In *Digital Avionics Systems Conference, 2008. DASC 2008. IEEE/AIAA 27th*, pages 1–E. IEEE, 2008.

[278] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, 2009.

[279] Andres J Ramirez and Betty HC Cheng. Design patterns for developing dynamically adaptive systems. In *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pages 49–58. ACM, 2010.

[280] Brian Randell, Pete Lee, and Philip C. Treleaven. Reliability issues in computing system design. *ACM Computing Surveys (CSUR)*, 10(2):123–165, 1978.

[281] Reinhard Reichel and Michael Armbruster. X-by-Wire Plattform-Konzept und Auslegung. *at-Automatisierungstechnik*, 59(9):583–597, 2011.

[282] Felix Reimann, Michael Glaß, Christian Haubelt, Michael Eberl, and Jürgen Teich. Improving platform-based system synthesis by satisfiability modulo theories solving. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 135–144. ACM, 2010.

[283] Leanna Rierson. *Developing safety-critical software: A practical guide for aviation software and DO-178C compliance*. CRC Press, 2013.

[284] Bill Ripley. Military COTS-based systems - Not necessarily right off the shelf. In *CompactPCI Systems*, 2004.

[285] G.D. Rodosek, K. Geihs, H. Schmeck, S. Burkhard, A. Andrzejak, K. Geihs, O. Shehory, and J. Wilkes. Self-healing systems: Foundations and challenges. In *Self-Healing and Self-Adaptive Systems, Germany. Dagstuhl Seminar Proceedings*, volume 09201. Dagstuhl 09201, Leibniz-Zentrum fuer Informatik, Germany, 2009.

[286] Oliver Rooks, Michael Armbruster, Armin Sulzmann, Gernot Spiegelberg, and Uwe Kiencke. Duo duplex drive-by-wire computer system. *Reliability Engineering & System Safety*, 89(1):71–80, 2005.

[287] Judith E. Y. Rossebeø, Mass Soldal Lund, Knut Eilif Husa, and Atle Refsdal. A conceptual model for service availability. In *Quality of Protection*, pages 107–118. Springer, 2006.

[288] RTCA. DO-178C, software considerations in airborne systems and equipment certification, 2011.

[289] Alejandra Ruiz, Garazi Juez, Philipp Schleiß, and Gereon Weiß. A safe generic adaptation mechanism for smart cars. In *IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, 2015.

[290] Alejandra Ruiz, Garazi Juez, Philipp Schleiss, and Gereon Weiss. A safe generic adaptation mechanism for smart cars. In *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*, pages 161–171. IEEE, 2015.

[291] V. Rupanov, C. Buckl, L. Fiege, M. Armbruster, A. Knoll, and G. Spiegelberg. Early safety evaluation of design decisions in e/e architecture according to iso 26262. In *Proceedings of the 3rd international ACM SIGSOFT symposium on Architecting Critical Systems*, pages 1–10. ACM, 2012.

[292] John Rushby. Partitioning in avionics architectures: Requirements, mechanisms, and assurance. Technical report, NASA Contractor Report CR-1999-209347; DOT/FAA/AR-99/58, 1999.

[293] John Rushby. Trustworthy self-integrating systems. In *International Conference on Distributed Computing and Internet Technology*, pages 19–29. Springer, 2016.

[294] John M Rushby. Design and verification of secure systems. *8th ACM Symposium on Operating System Principles*, 15(5):12–21, 1981.

[295] SAE. Architecture analysis and design language. `http://www.aadl.info`.

[296] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4(2):14, 2009.

[297] Mazeiar Salehie and Ladan Tahvildari. Towards a goal-driven approach to action selection in self-adaptive software. *Software: Practice and Experience*, 42(2):211–233, 2012.

[298] Soheil Samii, Unmesh D Bordoloi, Petru Eles, Zebo Peng, and Anton Cervin. Control-quality optimization for distributed embedded systems with adaptive fault tolerance. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 68–77. IEEE, 2012.

[299] Prabhat Kumar Saraswat, Paul Pop, and Jan Madsen. Task migration for fault-tolerance in mixed-criticality embedded systems. In *2nd International Workshop on Adaptive and Reconfigurable Embedded Systems (APRES)*, 2009.

[300] Titos Saridakis. Design patterns for graceful degradation. In *Transactions on Pattern Languages of Programming I (TPLOP)*, pages 67–93. Springer, 2009.

[301] Tripti Saxena. *A generic framework for design space exploration*. PhD thesis, Vanderbilt University, 2012.

[302] M. Norazizi Sham Mohd Sayuti and Leandro Soares Indrusiak. A function for hard real-time system search-based task mapping optimisation. In *IEEE 18th International Symposium on Real-Time Distributed Computing (ISORC)*, 2015.

[303] Bernhard Schaetz, Florian Holzl, and Torbjörn Lundkvist. Design-space exploration through constraint-based model-transformation. In *Engineering of Computer Based Systems (ECBS), 2010 17th IEEE International Conference and Workshops on*, pages 173–182. IEEE, 2010.

[304] Bernhard Schätz. Model-based development of software systems. Habilitation Thesis, Technical University Munich, 2009.

[305] Bernhard Schätz, Sebastian Voss, and Sergey Zverlov. Automating Design-Space Exploration: Optimal Deployment of Automotive SW-Components in an ISO26262 Context. In *Design Automation Conference (DAC)*, 2015.

[306] Hartmut Schmeck. Organic computing – a new vision for distributed embedded systems. In *Object-Oriented Real-Time Distributed Computing, 2005. ISORC 2005. Eighth IEEE International Symposium on*, pages 201–203. IEEE, 2005.

[307] Hartmut Schmeck, Christian Müller-Schloer, Emre Cakar, Moez Mnif, and Urban Richter. Adaptivity and self-organisation in organic computing systems. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 5(3):10:1–10:32, September 2010.

[308] Konstantin Schorp and Stephan Sommer. Component-Based Modeling and Integration of Automotive Application Architectures. In *IEEE International Electric Vehicle Conference (IEVC)*, pages 1–7. IEEE, 2014.

[309] H. Seebach, F. Nafz, J. Holtmann, J. Meyer, M. Tichy, W. Reif, and W. Schäfer. Designing self-healing in automotive systems. *Autonomic and Trusted Computing*, pages 47–61, 2010.

[310] G.D.M. Serugendo, J. Fitzgerald, A. Romanovsky, and N. Guelfi. Metaself-a framework for designing and controlling self-adaptive and self-organising systems. *School Comput. Sci. Inf. Syst., Birkbeck College, London, UK, Tech. Rep. BBKCS-08-08*, 2008.

[311] O. Shehory, J. Martinez, A. Andrzejak, C. Cappiello, W. Funika, D. Kondo, L. Mariani, B. Satzger, and M. Schmid. Self-healing and recovery methods and their classification. In *Self-Healing and Self-Adaptive Systems, Germany. Dagstuhl Seminar Proceedings*, volume 09201. Dagstuhl 09201, Leibniz-Zentrum fuer Informatik, Germany, 2009.

[312] Charles P Shelton and Philip Koopman. Using architectural properties to model and measure graceful degradation. In *Architecting dependable systems*, pages 267–289. Springer, 2003.

[313] Charles P Shelton, Philip Koopman, and William Nace. A framework for scalable analysis and design of system-wide graceful degradation in distributed embedded systems. In *Proceedings of the 8th Int. Workshop on Object-Oriented Real-Time Dependable Systems (WORDS)*, pages 156–163. IEEE, 2003.

[314] Charles Preston Shelton. *Scalable graceful degradation for distributed embedded systems*. PhD thesis, Carnegie Mellon University, 2003.

[315] Kang G Shin and Charles L Meissner. Adaptation and graceful degradation of control system performance by task reallocation and period adjustment. In *Real-Time Systems, 1999. Proceedings of the 11th Euromicro Conference on*, pages 29–36. IEEE, 1999.

[316] Jocelyn Simmonds and María Cecilia Bastarrica. Modeling variability in software process lines. *Departamento de Ciencias de la Computación. Universidad de Chile*, 2011.

[317] P. Sinha. Architectural design and reliability analysis of a fail-operational brake-by-wire system from iso 26262 perspectives. *Reliability Engineering & System Safety*, 2011.

[318] Stephan Sommer, Alexander Camek, Klaus Becker, Christian Buckl, Alois Knoll, Andreas Zirkler, Ludger Fiege, Michael Armbruster, and Gernot Spiegelberg. RACE: A Centralized Platform Computer Based Architecture for Automotive Applications. In *IEEE Vehicular Electronics Conference / Int. Electric Vehicle Conference (VEC-IEVC)*, pages 1–6. IEEE, 2013.

[319] Ioana Sora, Pierre Verbaeten, and Yolande Berbers. A description language for composable components. In *International Conference on Fundamental Approaches to Software Engineering*, pages 22–36. Springer, 2003.

[320] Cary R. Spitzer. *The Avionics Handbook*. CRC Press, 1st edition, 2001.

[321] Cary R. Spitzer, Uma Ferrell, and Thomas Ferrell. *Digital Avionics Handbook, Chapter 5, Fault-Tolerant Avionics (by Ellis F. Hitt)*. CRC Press, 3rd edition, 2014.

[322] Hauke Stahle, Ljubo Mercep, Alois Knoll, and Gernot Spiegelberg. Towards the deployment of a centralized ICT architecture in the automotive domain. In *Embedded Computing (MECO), 2013 2nd Mediterranean Conference on*, pages 66–69. IEEE, 2013.

[323] S. Stein, M. Neukirchner, and R. Ernst. Admission control and self-configuration in the epoc framework. In *Embedded Computer Systems (SAMOS), 2011 International Conference on*, pages 364–371. IEEE, 2011.

[324] Steffen Stein. *Allowing Flexibility in Critical Systems: The EPOC Framework*. PhD thesis, TU Braunschweig, Germany, 2012.

[325] T. Streichert, C. Haubelt, D. Koch, and J. Teich. Concepts for self-adaptive and self-healing networked embedded systems. *Organic Computing*, pages 241–260, 2008.

[326] J. Sudeikat, J.P. Steghöfer, H. Seebach, W. Reif, W. Renz, T. Preisler, and P. Salchow. On the combination of top-down and bottom-up methodologies for the design of coordination mechanisms in self-organising systems. *Information and Software Technology*, 54(6):593–607, 2012.

[327] Ivan Švogor, Ivica Crnkovic, and Neven Vrcek. An extended model for multi-criteria software component allocation on a heterogeneous embedded platform. *CIT. Journal of Computing and Information Technology*, 21(4):211–222, 2014.

[328] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component software: beyond object-oriented programming*. Addison-Wesley, ACM Press New York, 2 edition, 2002.

[329] E-G Talbi and Traian Muntean. Hill-climbing, simulated annealing and genetic algorithms: a comparative study and application to the mapping problem. In *System Sciences, 1993, Proceeding of the Twenty-Sixth Hawaii International Conference on*, volume 2, pages 565–573. IEEE, 1993.

[330] Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy. *Software architecture: foundations, theory, and practice*. Wiley, 2009.

[331] Christopher Temple and Antonio Vilela. Safety Automotive - Fehlertolerante Systeme im Fahrzeug – von "fail-safe" zu "fail-operational". `http://www.elektroniknet.de/automotive/assistenzsysteme/artikel/110612`, 2014.

[332] Abhilash Thekkilakattil, Radu Dobrin, and Sasikumar Punnekkat. Mixed criticality scheduling in fault-tolerant distributed real-time systems. In *Embedded Systems (ICES), 2014 International Conference on*, pages 92–97. IEEE, 2014.

[333] Ken W Tindell, Alan Burns, and Andy J. Wellings. Allocating hard real-time tasks: an np-hard problem made easy. *Real-Time Systems*, 4(2):145–165, 1992.

[334] M. Trapp, R. Adler, M. Förster, and J. Junger. Runtime adaptation in safety-critical automotive systems. In *Proceedings of the 25th conference on IASTED International Multi-Conference: Software Engineering*, pages 308–315. ACTA Press, 2007.

[335] Mario Trapp. *Modeling the adaptation behavior of adaptive embedded systems*. PhD thesis, TU Kaiserslautern, 2005.

[336] Mario Trapp, Daniel Schneider, and Peter Liggesmeyer. A safety roadmap to cyber-physical systems. In *Perspectives on the Future of Software Engineering*, pages 81–94. Springer, 2013.

[337] Matthias Traub. *Durchgängige Timing-Bewertung von Vernetzungsarchitekturen und Gateway-Systemen im Kraftfahrzeug*. KIT Scientific Publishing, 2010.

[338] Pascal Traverse, Christine Bezard, Jean-Michel Camus, Isabelle Lacaze, Hervé Leberre, Patrick Ringeard, and Jean Souyris. Dependable avionics architectures: example of a fly-by-wire system. *Safety of Computer Architectures*, pages 199–232, 2013.

[339] Pascal Traverse, Isabelle Lacaze, and Jean Souyris. A process toward total dependability-airbus fly-by-wire paradigm. In *EDCC*, page 1. Springer, 2005.

[340] Andreas Vogelsang. *Model-based Requirements Engineering for Multifunctional Systems*. PhD thesis, Technical University of Munich, 2015.

[341] Andreas Vogelsang, Sebastian Eder, Georg Hackenberg, Maximilian Junker, and Sabine Teufl. Supporting concurrent development of requirements and architecture: A model-based approach. In *2nd Int. Conf. on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 587–595. IEEE, 2014.

[342] Andreas Vogelsang and Steffen Fuhrmann. Why feature dependencies challenge the requirements engineering of automotive systems: An empirical study. In *21st IEEE Int. Requirements Engineering Conference (RE)*, pages 267–272. IEEE, 2013.

[343] Andreas Vogelsang, Stefan Teuchert, and Jean-François Girard. Extent and characteristics of dependencies between vehicle functions in automotive software systems. In *Proceedings of the 4th International Workshop on Modeling in Software Engineering*, pages 8–14. IEEE Press, 2012.

[344] Andrija Volkanovski, Marko Čepin, and Borut Mavko. Application of the fault tree analysis for assessment of power system reliability. *Reliability Engineering & System Safety*, 94(6):1116–1127, 2009.

[345] Sebastian Voss, Johannes Eder, and Florian Hölzl. Design space exploration and its visualization in autofocus3. In *Software Engineering Workshops 2014, GI, Kiel*, pages 57–66, 2014.

[346] Sebastian Voss, Johannes Eder, and Bernhard Schätz. Schedule synthesis for multi-period sw components. In *SAE World Congress*, 2016.

[347] Sebastian Voss and Bernhard Schätz. Deployment and scheduling synthesis for mixed-critical shared memory applications. In *Engineering of Computer Based Systems (ECBS)*, 2013.

[348] Sebastian Voss and Bernhard Schätz. Asil-conformant deployment and schedule synthesis using multi-objective design space exploration. In *Keynote at Workshop on TIming Performance in Safety Engineering (TIPS) at International Conference on Computer Safety, Reliability, and Security (SAFECOMP)*, volume 9923, page 393. Springer, 2016.

[349] Florian Wagner, Fuyuki Ishikawa, and Shinichi Honiden. Robust service compositions with functional and location diversity. *IEEE Transactions on Services Computing*, 9(2):277 – 290, 2016.

[350] Stefan Wagner. *Software product quality control*. Springer, 2013.

[351] Stefan Wagner et al. *Cost optimisation of analytical software quality assurance.* PhD thesis, Technical University Munich, 2007.

[352] Peng Wang, Kaiyuan Zhang, Rong Chen, Haibo Chen, and Haibing Guan. Replication-based fault-tolerance for large-scale graph processing. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 562–573. IEEE, 2014.

[353] Christopher B Watkins and Randy Walter. Transitioning from federated avionics architectures to integrated modular avionics. In *Digital Avionics Systems Conference, 2007. DASC'07. IEEE/AIAA 26th*, pages 2–A. IEEE, 2007.

[354] Raphael Weber, Stefan Henkler, and Achim Rettberg. Multi–objective design space exploration for cyber–physical systems satisfying hard real–time and reliability constraints. In *Proceedings of IDEAL'14 Workshop*, IFIP Springer Series. CPSWeek 2014 – IDEAL'14 Workshop, Springer, 2014.

[355] Junqing Wei, Jarrod M Snider, Junsung Kim, John M Dolan, Raj Rajkumar, and Bakhtiar Litkouhi. Towards a viable autonomous driving research platform. In *Intelligent Vehicles Symposium (IV), 2013 IEEE*, pages 763–770. IEEE, 2013.

[356] James Windsor and Kjeld Hjortnaes. Time and space partitioning in spacecraft avionics. In *Space Mission Challenges for Information Technology, 2009. SMC-IT 2009. Third IEEE International Conference on*, pages 13–20. IEEE, 2009.

[357] Ernest Woźniak. *Model-based Synthesis of Distributed Real-time Automotive Architectures*. PhD thesis, Université Paris Sud XI, 2014.

[358] YC Bob Yeh. Triple-triple redundant 777 primary flight computer. In *Aerospace Applications Conference, 1996. Proceedings., 1996 IEEE*, volume 1, pages 293–307. IEEE, 1996.

[359] YC Bob Yeh. Safety critical avionics for the 777 primary flight controls system. In *Digital Avionics Systems, 2001. DASC. 20th Conference*, volume 1, pages 1C2–1. IEEE, 2001.

[360] Ying C Yeh. Design considerations in boeing 777 fly-by-wire computers. In *High-Assurance Systems Engineering Symposium, 1998. Proceedings. Third IEEE International*, pages 64–72. IEEE, 1998.

[361] M. Zeller, C. Prehofer, G. Weiss, D. Eilers, and R. Knorr. Towards self-adaptation in real-time, networked systems: Efficient solving of system constraints for automotive embedded systems. In *5th IEEE Int. Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, pages 79–88. IEEE, 2011.

[362] Marc Zeller and Christian Prehofer. Modeling and efficient solving of extra-functional properties for adaptation in networked embedded real-time systems. *Journal of Systems Architecture (JSA)*, 59(10):1067–1082, 2013.

[363] Haibo Zeng, Abhijit Davare, Alberto Sangiovanni-Vincentelli, Sampada Sonalkar, Sri Kanajan, and Claudio Pinello. Design space exploration of automotive platforms in metropolis. Technical report, SAE Technical Paper, 2006.

[364] Qi Zhu, Haibo Zeng, Wei Zheng, Marco DI Natale, and Alberto Sangiovanni-Vincentelli. Optimization of task allocation and priority assignment in hard real-time distributed systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 11(4):85, 2012.

[365] Bastian Zimmer, Susanne Bürklen, Jens Höfflinger, Mario Trapp, and Peter Liggesmeyer. Safety-focused deployment optimization in open integrated architectures. In *Computer Safety, Reliability, and Security*, pages 328–339. Springer, 2012.

[366] Werner Zimmermann and Ralf Schmidgall. *Bussysteme in der Fahrzeugtechnik*. Springer, 5th edition, 2014.

[367] Glenn Zorpette. Computers that are 'never' down: Fault-tolerant systems based on multiple microprocessors trade off throughput for enhanced reliability. *Spectrum, IEEE*, 22(4):46–54, 1985.

[368] Sergey Zverlov, Maged Khalil, and Mayank Chaudhary. Pareto-efficient deployment synthesis for safety-critical applications in seamless model-based development. In *Embedded Real-Time Software and Systems (ERTS2)*, 2016.