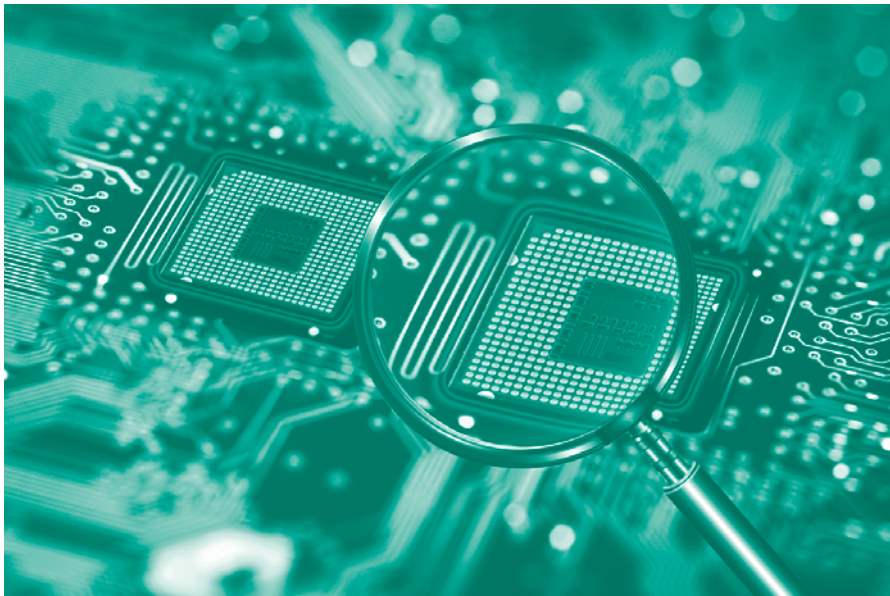


Bastian Zimmer

Efficiently Deploying Safety-Critical Applications onto Open Integrated Architectures



Editor-in-Chief: Prof. Dr. Dieter Rombach
Editorial Board: Prof. Dr. Frank Bomarius
Prof. Dr. Peter Liggesmeyer
Prof. Dr. Dieter Rombach

FRAUNHOFER VERLAG

PhD Theses in Experimental Software Engineering

Volume 50

Editor-in-Chief: Prof. Dr. Dieter Rombach

Editorial Board: Prof. Dr. Frank Bomarius
Prof. Dr. Peter Liggesmeyer
Prof. Dr. Dieter Rombach

Zugl.: Kaiserslautern, Univ., Diss., 2014

Printing:
Mediendienstleistungen des
Fraunhofer-Informationszentrum Raum und Bau IRB, Stuttgart

Printed on acid-free and chlorine-free bleached paper.

All rights reserved; no part of this publication may be translated, reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. The quotation of those designations in whatever way does not imply the conclusion that the use of those designations is legal without the consent of the owner of the trademark.

© by **Fraunhofer Verlag**, 2014
ISBN (Print): 978-3-8396-0753-4
Fraunhofer-Informationszentrum Raum und Bau IRB
Postfach 800469, 70504 Stuttgart
Nobelstraße 12, 70569 Stuttgart
Telefon +49 711 970-2500
Telefax +49 711 970-2508
E-Mail verlag@fraunhofer.de
URL <http://verlag.fraunhofer.de>

Efficiently Deploying Safety-Critical Applications onto Open Integrated Architectures

Vom Fachbereich Informatik
der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades

Doktor der Ingenieurwissenschaften (Dr.-Ing.)

genehmigte Dissertation
von

Dipl.-Inf. Bastian Zimmer

Fraunhofer-Institut für Experimentelles Software Engineering
(Fraunhofer IESE)
Kaiserslautern

Berichterstatter:

Prof. Dr.-Ing. Peter Liggesmeyer
Prof. Dr.rer.nat. Karsten Berns

Dekan:

Prof. Dr.rer.nat. Klaus Schneider

Tag der Wissenschaftlichen Aussprache:

21.02.2014

D 386

For my parents

Abstract

Open integrated architectures such as AUTOSAR or IMA offer an increased modularity and flexibility over more established federated architectures. Using such a design, system developers can reuse and exchange applications and execution platforms more flexibly, as costs for migration and integration decrease. However, when developing systems that are safety-critical, the traditionally monolithic approach of safety engineering poses threats to the modularity that comes with the new architecture. In fact, the safety has to be re-evaluated and argued whenever the system changes. As a consequence, significant costs are incurred every time a component is reused or replaced, which decreases the desired flexibility of the open integrated architecture.

To address this problem, this thesis introduces a technique that allows for the partial automation of the safety-related integration process. The technique is built of three components:

The foundation of our approach is a model-based specification language allowing developers to define the conditions for the valid integration of platforms and applications. Our language follows a modular, contract-based approach for the specification of demands and guarantees, which together form a safety interface between application and platform. The demands are specified by the application developer and define the safety-related behavior of the platform as required for the safe execution of the application. The guarantees, on the other hand, are specified by the platform developer and define the actual safety-related capabilities of the platform at hand.

Based on this language, we define a mediation algorithm that is capable of automatically checking if the conditions specified in the safety interfaces are met for a given application-platform deployment. This automation decreases the effort for integrating safety-critical applications and platforms, which sustains the flexibility of the design.

However, in order to perform the automated integration check, our mediation algorithm requires the deployment of applications and platforms as an input. To assist the integrator in identifying a valid deployment, we present an objective function for evaluating safety-related deployment criteria as a third and final component of our solution approach.

Acknowledgements

I would like to thank my supervisor Prof. Dr.-Ing. Peter Liggesmeyer for his counsel and support and the chance to obtain my doctorate at his working group.

I am especially thankful towards my advisor Mario Trapp. His advice, encouragement and overall support, from the very beginning to the very end of my research, have been invaluable.

I would also like to thank my industrial colleagues at Bosch, Susanne Bürklen and Jens Höfflinger, for their advice during the early phases of my doctorate and their help with joint publications.

Last but not least, I would like to thank my current and former co-workers and friends at Fraunhofer IESE, for their helpful comments and productive discussions.

Table of Contents

1	Introduction	1
1.1	Context and Motivation	1
1.2	Problem Statement and Contribution	4
1.3	Structure	6
2	Related Work	7
2.1	Open Integrated Architectures	7
2.1.1	Platform-based Design	10
2.1.2	Example Platform 1: AUTOSAR	14
2.1.3	Example Platform 2: IMA – ARINC 653	17
2.1.4	Platform Service Types	19
2.2	Deployment Evaluation	22
2.2.1	Match	26
2.2.2	Replication and Reliability	27
2.2.3	Delay and Flow	29
2.2.4	Fixed and Diverse Assignment	30
2.2.5	Mixed Criticality	31
2.3	Modular Certification	32
2.3.1	Process-focused Approaches	37
2.3.2	Modular GSN	40
2.3.3	Rich Component Model	41
2.3.4	Safety Analysis of CRMS	41
2.3.5	Conditional Safety Certificates	43
3	Solution Overview	45
3.1	Interface Specification and Mediation	48
3.2	Deployment Evaluation	50
4	Interface Specification	53
4.1	Running Example	55
4.2	Language Design	58
4.3	Common Language – General Features	61
4.3.1	Demands and Guarantees	62
4.3.2	Conditions	63
4.3.3	Architecture Relations	65
4.3.4	Parameters	69
4.4	Common Language – Failures and Failure Reactions	77
4.4.1	Failure Analysis and Classification	78
4.4.2	Platform Service Failures	81
4.4.3	Application Failures	109
4.4.4	Platform Failure Reactions	113
4.5	Application Language	116

4.5.1	Platform Service Demands.....	118
4.5.2	Health Monitoring Demands.....	123
4.5.3	Resource Protection Demands.....	128
4.5.4	Service Diversity Demands.....	131
4.6	Platform Language.....	134
4.6.1	Platform Service Guarantees	136
4.6.2	Health Monitoring Guarantees.....	141
4.6.3	Resource Protection Guarantees	147
4.6.4	Service Diversity Guarantees.....	149
5	Interface Mediation.....	153
5.1	Configuration.....	155
5.1.1	Platform Object Instantiation.....	156
5.1.2	Evaluation of Configuration-Dependent Conditions	157
5.2	System Integration	158
5.2.1	Evaluation of Deployment-Dependent Conditions.....	159
5.3	Manual Condition Evaluation	160
5.4	Interface Mediation	160
5.4.1	Mediation of Platform Service Demands.....	162
5.4.2	Mediation of Application Monitoring Demands.....	169
5.4.3	Mediation of Failure Reaction Demands	173
5.4.4	Mediation of Resource Protection Demands.....	178
5.4.5	Mediation of Service Diversity Demands.....	183
5.4.6	Mediation of Specific Parameters	185
5.5	Post Mediation	188
5.5.1	Visualizing Mediation Results	188
6	Deployment Evaluation	191
6.1	Problem Statement	192
6.1.1	Safety-Related Properties	194
6.1.2	Running Example	195
6.2	Objective Function.....	196
6.2.1	Cohesion Metric	196
6.2.2	Coupling Metric.....	198
6.2.3	Constraints	200
6.2.4	Objective Function Assembly.....	201
6.2.5	Parameterization	201
6.3	Deployment Optimization	203
7	Implementation and Evaluation	207
7.1	VerSal Evaluation.....	207
7.2	Deployment Evaluation	214
7.3	VerSal Implementation	217
7.4	Deployment Implementation	220
8	Conclusion.....	223
8.1	Contributions and Limitations	223
8.2	Future Work	225

8.3 Final Comment	226
9 References.....	227
Appendix A Architectural Meta-Model	237
A.1 System Meta-Model	238
A.2 Application Meta-Model	240
A.3 Platform Meta-Model.....	247
A.4 Deployment Meta.Model	252
Appendix B List of Common Failure Modes and Reactions	254
Appendix C NLR using EBNFs	258

List of Figures

Figure 1: Schematic of an open integrated architecture	2
Figure 2: Simplified example a platform software (PSW) architecture..	10
Figure 3: A development process for open integrated systems.....	11
Figure 4: Mapping of a functional architecture onto a platform topology	12
Figure 5: Interactions between abstraction layers in a platform-based design.....	13
Figure 6: AUTOSAR methodology excerpt.....	15
Figure 7: The layered platform software architecture of AUTOSAR	16
Figure 8: ARINC 653 software architecture and its relations.....	18
Figure 9: A classification of the different aspects of modular certification.....	33
Figure 10: Dependencies between different kinds of evidences	34
Figure 11: A more detailed version of the OIA development process	46
Figure 12: Overall process of interface specification and interface mediation	48
Figure 13: The process of Solution Candidate Identification.....	51
Figure 14: The role of the VerSal language in the VerSal method.....	54
Figure 15: This figure shows the top-level architecture of the VerSal language.....	54
Figure 16: An example cruise control application	55
Figure 17: An example platform	56
Figure 18: The four classes of safety dependencies in VerSal.....	59
Figure 19: The high-level architecture of the VerSal language.....	61
Figure 20: An excerpt of the VerSal language's classification tree	63
Figure 21: The VerSal condition state machine	64
Figure 22: The VerSal condition meta-model	65
Figure 23: Containment relations in VerSal.....	67
Figure 24: Reference relations in VerSal	68
Figure 25: Parameter assignments	70
Figure 26: The primitive data types of the VerSal language.....	71
Figure 27: The meta-model of the composite time parameter.....	72
Figure 28: The meta-model of the time constraint classes.....	73
Figure 29: The meta-model of the physical quantity parameter.....	74
Figure 30: The meta-model of the composite error parameter	75
Figure 31: The meta-model of the integrity level parameter.....	76
Figure 32: The assignment of the integrity level parameter	77

Figure 33: The failure mode taxonomy used in VerSal.....	78
Figure 34: The state machine of a mutex.....	83
Figure 35: The meta-model of the synchronization failure model.....	84
Figure 36: Comparison of functional and technical communication scenarios.....	87
Figure 37: The meta-model of the communication failure model.....	87
Figure 38: The different scenarios for accessing an input channel.....	91
Figure 39: The meta-model of the analog part of the input failure model.....	93
Figure 40: The scenario for accessing an output peripheral.....	98
Figure 41: The meta-model of the output failure model.....	99
Figure 42: The scenarios for using time-related services.....	101
Figure 43: The meta-model of the time services failure model.....	102
Figure 44: The scenario for accessing indirectly accessed memory.....	103
Figure 45: The meta-model of the memory service failure model.....	104
Figure 46: Events and intervals for task scheduling.....	106
Figure 47: The meta-model of the scheduling failure model.....	107
Figure 48: The meta-model of the basic computation failure model....	109
Figure 49: An excerpt of the meta-model for application monitoring demands.....	111
Figure 50: An excerpt of the platform failure reaction meta-model.....	114
Figure 51: The top-level meta-model of the application language.....	117
Figure 52: Flow chart showing the service failure demand design options.....	119
Figure 53: The principal structure of the platform service demand meta-model.....	121
Figure 54: The health monitoring meta-model.....	124
Figure 55: An excerpt of the resource protection meta-model.....	130
Figure 56: The service diversity meta-model.....	133
Figure 57: The top-level meta-model of the platform language.....	136
Figure 58: The top-level structure of the platform service guarantee meta-model.....	138
Figure 59: The health monitoring guarantee meta-model.....	141
Figure 60: An overview of the resource protection guarantee model..	148
Figure 61: The service diversity guarantee meta-model.....	151
Figure 62: An overview of the VerSal mediation.....	154
Figure 63: The process of automatic platform object guarantee generation.....	157
Figure 64: An overview of the automatic demand mediation process..	162
Figure 65: An overview of platform service demand mediation.....	164
Figure 66: The concept of related guarantees.....	166

Figure 67: An overview of the application monitoring demand mediation	171
Figure 68: An overview of failure reaction demand mediation	175
Figure 69: An overview of resource protection demand mediation	180
Figure 70: An overview of service diversity demand mediation.....	185
Figure 71: The structure of a mediation report.....	189
Figure 72: An overview of our deployment evaluation contribution	192
Figure 73: The meta-model to specify deployment problems	193
Figure 74: A running example for the deployment evaluation method	195
Figure 75: Two example deployments illustrating the cohesion metric	198
Figure 76: Two example deployments illustrating the coupling metric.	200
Figure 77: The tool-supported parameterization of the objective function	203
Figure 78: The adapted GA optimization loop used for our objective function	204
Figure 79: The open-loop evaluation of the objective function.....	215
Figure 80: The closed-loop evaluation of the objective function.....	217
Figure 81: An overview of the VerSal editors	219
Figure 82: The system editor UI for controlling the mediation	220
Figure 83: The mediation report viewer	220
Figure 84: The tool chain used by our deployment optimization prototype.....	221
Figure 85: A GMF based visualization of a solved deployment problem	222
Figure 86: Relations between the VerSal language and the architecture model.....	237
Figure 87: The system meta-model	238
Figure 88: The top-level application meta-model	241
Figure 89: Application meta-model with a focus on the ASWC element	243
Figure 90: Application meta-model with a focus on the ServiceNeed element	245
Figure 91: Application meta-model with a focus on the Port element.	246
Figure 92: The top-level platform meta-model	248
Figure 93: Platform meta-model with a focus on the Device element..	250
Figure 94: Platform meta-model with a focus on the Service element.	252
Figure 95: Excerpt of the deployment meta-mode	253

List of Tables

Table 1:	Quality criteria of a given deployment solution.....	25
Table 2:	A mapping between quality criteria and references.....	25
Table 3:	A mapping of identifiers used in Table 2 to references.	26
Table 4:	An overview of the modular certification aspects	36
Table 5:	Mapping of identifiers used in Table 4 to references.	36
Table 6:	The deployment of the running example.....	57
Table 7:	A list of all platform service demands.....	121
Table 8:	A list of all platform service demands.....	139
Table 9:	A list of all platform service demands.....	145
Table 10:	A list of parameters of our objective function.....	202
Table 11:	Mapping of AUTOSAR services to services in VerSal	208
Table 12:	Mapping of ARINC 653 services to services in VerSal.....	209
Table 13:	Mapping of monitoring mechanisms to the VerSal language.....	210
Table 14:	Mapping of platform failure reactions to the VerSal language.....	211
Table 15:	An overview of the VerSal testing	213
Table 16:	All resource mappings specified in the deployment model .	252
Table 17:	Failure modes specified in the common language	254
Table 18:	Failure reaction specified in the common language	257
Table 19:	The notational elements of the EBNF used in this document	258

1 Introduction

1.1 Context and Motivation

More and more developers of distributed embedded systems choose to design their product in accordance with the principles of *integrated architectures*. Unlike *federated architectures*, where every function is executed on a dedicated isolated computer platform, integrated architectures allow multiple applications to share the same execution node and to communicate with each other via shared communication networks. As a consequence, functions can be integrated more tightly and computer platforms can be saved. This integration allows for cost and weight improvements, at the expense of fault isolation and error containment [1].

The development of standards like *ARINC 653* [2] and *AUTOSAR* [3] have shown that the advantages of integrated architectures are further enriched when there is a public and standardized interface between application and execution platform. Architectures with a public standardized Application Programming Interface (API) are often labeled as open, since third-party manufacturers can develop compatible components and contribute to the architecture. In the case of such an *open integrated architecture*, separate manufacturers are able to develop compatible *applications* and *platforms* and, ideally, the organization responsible for system integration is capable of fitting the emerging *open integrated system* together with little effort.

The API separates function-specific applications and general-purpose execution platforms, and is realized by operating systems, drivers, or more generally, by *platform software* (PSW) or middleware. The middleware abstracts from the specifics of the underlying *platform hardware* and provides a uniform interface to allow applications to access the platform's shared resources. Furthermore, the middleware acts as a resource manager, allowing for a well-organized distribution of resources. Figure 1 sketches the characteristics of open integrated architectures described above.

This design has many advantages: First, it allows for a clear separation between functional and technical design and thus provides a means for abstraction and complexity management. Second, the roles of

application developer, platform developer¹, and system integrator can more easily be filled by different organizations, since the standardized API fosters modularized development. This facilitates modular contracting, which is beneficial for the distributed development of complex embedded systems. Finally, the standardized API makes applications as well as platforms more portable and exchangeable, allowing the integrator to use them more flexibly.

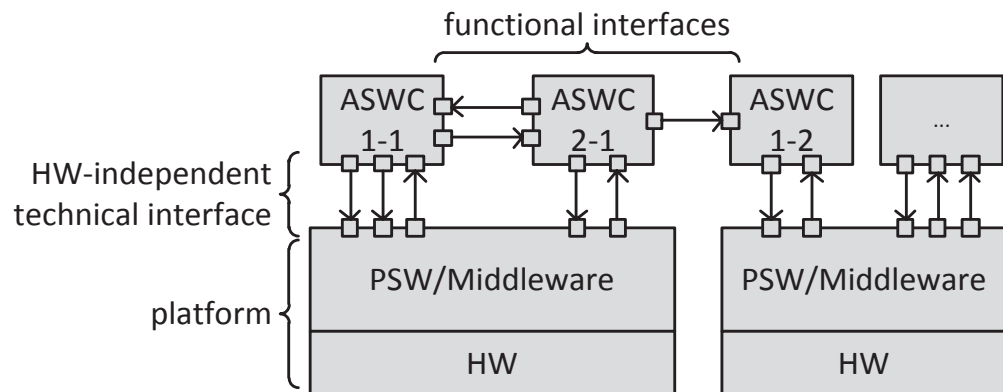


Figure 1: Schematic of an open integrated architecture, showing several application software components (ASWC) and two platforms. Each platform consists of middleware and hardware (HW).

The aforementioned integrators benefit from this flexibility in multiple ways. First, they are able to reuse established platforms and applications in new systems, without having to spend an extensive amount of resources on adaptation [4]. Second, they can, over the lifetime of a system, exchange execution platforms to fight hardware obsolescence, or integrate novel applications to expand or modify the system's functionality. Finally, system integrators are able to freely choose efficient deployment of the system's *functional architecture*, containing several *application software components* (ASWC), onto the system's *platform topology*, consisting of execution platforms and communication networks.

The integrator can optimize the allocation of ASWCs with regard to multiple aspects. The most common optimization criterion is to closely match the requirements of the applications to the capabilities of the available platforms. This has the potential of reducing the number of required platforms and thus to reduce hardware costs as well as weight, space, and energy consumption. Yet, finding a beneficial deployment is challenging for at least two reasons. First, the integrator has to consider not only a multitude of criteria, but some of them are also conflicting in nature, like the satisfaction of response time constraints and the

¹ Often, platform development can be sub-divided into middleware and hardware development. This thesis, however, regards the platform as a whole.

maximization of resource utilization. Second, the integrator is confronted with a large solution space containing at least p^n possible deployments, assuming that there are p platforms and n applications.

The task of deployment becomes even more challenging if the system is safety-critical. A system is called safety-critical if its malfunction/failure may result in death or injury. Because of this, the development of safety-critical systems is typically regulated by standards and norms, and in some industries, standard compliance is checked by official certification bodies. The enormous responsibility towards society and the resulting strict development requirements make the general development of a safety-critical system a challenging and, typically, expensive endeavor.

As far as deployment is concerned, we can regard safety as a source of additional optimization criteria and constraints. The integrator, for example, has to check whether a platform fulfills all safety-related demands of the hosted applications and thus allows for safe execution. Furthermore, platforms may introduce new sources of common cause failure that may invalidate complete safety concepts or increase the criticality classification of software-components. Consequently, the safety aspect has to be taken into careful consideration in order not to endanger the overall safety of the system or inflict additional costs.

Additionally, safety criticality poses a threat to the portability of an open integrated system. In fact, the technical compatibility of applications and platforms provided by the API does not guarantee per se the feasibility of every possible platform-application combination. Especially in the context of safety-critical systems, there has to be a rigorous check whether each application software component is able to run safely on its dedicated platform. Additionally, safety standards often demand the creation of a seamless argument to demonstrate that a system is acceptably safe. In sum, checking safety and creating the evidences leads to effort that has to be spent every time an application is deployed to a different platform, which reduces the desired flexibility significantly.

Therefore, this thesis introduces a technique for efficiently checking and arguing the safety compatibility of an application-platform combination, so as to decrease the integration costs, regain portability of applications, and maintain the architecture's flexibility. Based on the regained flexibility, this thesis describes techniques for evaluating and optimizing a given deployment with respect to safety in order to aid the integrator in finding an optimal deployment and exploiting the advantages of an open integrated architecture.

1.2 Problem Statement and Contribution

Usually, safety engineering starts with the identification and classification of potential sources of harm posed by a system, which are called *hazards*. Hazard analysis is often performed separately for each function of the system, rather than for the system as a whole. Since in an integrated architecture, functions are implemented by applications, hazards can be identified for each application.

After hazard analysis, the safety engineer needs to identify all potential causes that may lead to the previously identified hazards and develop a strategy for controlling these inadvertent events. Since the application depends on other system components, it is typically unable to implement such a strategy in a completely self-supported way. For this reason, the implementation of the strategy depends on the behavior of other system elements. When examining an application in an open integrated architecture, one can differentiate three classes of dependencies.

The first class contains dependencies among multiple applications. We shall refer to this class as *functional* or *horizontal* dependency, in accordance with the structure outlined in Figure 1. In order to exemplify functional dependencies, let's assume a car's ESC² application requires knowing the precise vehicle speed to perform safely. Let's further assume that this information is provided by another application and can be used for the ESC's purposes. However, the ESC needs the other application to safely indicate when a precise vehicle speed is unavailable in order to be able to react accordingly and, e.g., deactivate the ESC to control the potential hazard.

We call the second class of dependencies *environmental*, as it subsumes all dependencies between the application and entities outside the E/E (electric/electronic) domain. Take, as an example, the time that the controlled system is able to tolerate a specific failure. In the case of the aforementioned ESC, the fault tolerance time between a faulty braking intervention and the hazardous destabilization of the car depends on the dynamics of each specific vehicle. In this case, the fault tolerance time might increase if the inertia of the vehicle increases.

The final class of dependencies is called *vertical* and contains the dependencies between applications and platforms. A platform can act both as a source of failure, for example when corrupting stored data, and as a provider of mechanisms for controlling failures, for example when detecting a crashed application through deadline monitoring. This thesis focuses on vertical dependencies.

² Electronic stability control (ESC) is a technology that improves the stability of a car by detecting and controlling skids.

The vertical dependencies have to be analyzed when checking the safety compatibility of an application and a platform. Since in an open integrated system, applications and platforms are developed independently, vertical dependencies have to be specified modularly. The resulting two elements of the modular specification are first, the demands of the application regarding the safety-related behavior of the platform and second, the guarantees regarding the actual capabilities of the platform. The vertical demands defined by the application developer should contain requirements such that the fulfillment of these requirements entails the platform behavior necessary for controlling the application's hazards. On the other hand, the guarantees specified by the platform developer should contain all safety-related capabilities of the platform, which can potentially be of use for an application. As a consequence, checking the safety compatibility between an application and a platform is a matter of checking whether the application demands can be fulfilled with the actual guarantees of the host platform.

In our experience, performing such a check manually is time-consuming and expensive, leading to the predicaments described in section 1.1. Because of this, it is necessary to automate this process as far as possible. However, in order to be able to do so, *application demands* and *platform guarantees* have to be formalized. Therefore, the first contribution of this thesis is defined as follows:

Contrib. 1 **Interface Specification:** *Defining a formal language for the modular specification of safety-related demands and guarantees between an application and a platform in an open integrated architecture.*

The ability to formally specify the vertical dependencies fulfills a prerequisite for analyzing safety compatibility in an automated way. However, since most safety standards demand a final safety assessment, it is also necessary that the results of such an analysis can be used as evidence to demonstrate that the system is acceptably safe. The generation of evidences and the demonstration of safety is a difficult and time-consuming task as well. Accordingly, the second contribution of this thesis addresses the following research problem:

Contrib. 2 **Interface Mediation:** *Developing an automated process for checking the safety compatibility of an application and a platform in an open integrated architecture.*

We believe that this automated process lowers the costs that accrue when deploying an application to a new platform and therefore enables improved deployment flexibility. To optimally use this flexibility, an appropriate deployment has to be identified. There are several safety-related aspects in a multi-criteria deployment optimization. In order to

take the last step towards capitalizing on flexible deployment, this thesis addresses the following problem:

Contrib. 3 **Deployment Evaluation:** *Developing an objective function for evaluating and optimizing the deployment of a functional architecture onto a platform topology from a safety perspective.*

This thesis explains the implementation of these three contributions and demonstrates how our solutions can be used to increase the efficiency of deploying safety-critical applications onto open integrated architectures.

1.3 Structure

In this chapter, we motivated the challenge of efficiently deploying safety-critical applications onto open integrated architectures and presented the research contributions proposed by this thesis, which aim at increasing the aforementioned efficiency. Chapter 2 presents an overview of related work in this field, so as to name the methods and techniques our approach is based upon and to sketch the knowledge gaps our approach tries to fill. Chapter 3 presents an overview of our solution and briefly outlines the structure of our methods and techniques. The following three chapters give an in-depth presentation of our solutions, structured according to the three contributions specified in section 1.2. Chapter 4 presents a language for specifying the safety-related dependencies between applications and platform, whereas chapter 5 describes a method for mediating those dependencies and, if possible, arguing how the specified constraints are met. Chapter 6 presents two metrics for a safety-focused deployment evaluation. The tools we implemented to evaluate our approach together with the evaluation itself are shown in chapter 7. We conclude and present possible future work in chapter 8.

2 Related Work

The previous chapter briefly introduced the challenges and the benefits of efficiently deploying safety-critical applications onto open integrated architectures and listed the contributions that will be presented in this thesis. This chapter presents related work in the area of open integrated architectures and deployment evaluation in order to explain the foundations our work is based upon and to precisely identify the knowledge gaps our methods and techniques try to fill.

Therefore, section 2.1 introduces open integrated architectures and the adaptation of the typical embedded system design process that they entail, and derives a common set of services provided by typical platforms using two popular examples. Section 2.2 focuses on the topic of deployment evaluation. We list several criteria that can be used to evaluate a deployment solution and present corresponding approaches to assess these criteria. Additionally, section 2.2 introduces the three criteria that are addressed by our techniques. Two of these criteria are assessed by our *objective functions* (see Contrib. 3) and the remaining criterion is the one that can be formulated with our specification language (see Contrib. 1) and checked with our mediation technique (see Contrib. 2). The related work regarding this criterion of checking whether the safety requirements of an application are fulfilled by the capabilities of its host platform is introduced in section 2.3. Since this last aspect evolves around the modular specification and integration of safety capabilities, we will use the commonly used term *modular certification* to headline this section.

2.1 Open Integrated Architectures

This section introduces the notion of open integrated architectures, starting with an explanation of the more traditional federating architectures and a presentation of the transitional solution of integrated architectures before ending with an explanation of open integrated architectures. After presenting the main characteristics of an open integrated architecture, we will explain the necessary adaptations of the traditional development lifecycle when developing open integrated systems. After that we will introduce the two most important standards for open integrated architectures before concluding this section with an independent list of platform services derived from the example standards.

In the past, the majority of distributed embedded systems were federated. [5] and [1] describe a *federated architecture* as a collection of computational nodes, where each node implements exactly one dedicated function, such as controlling the air speed of an aircraft (auto-throttle function) or the speed of a car (cruise-control function). Sensors and effectors were typically not shared and communication between the nodes, and thus the functions, was limited [6].

The following quote by Scott Gravelie, Director, Boeing 787 Programs with GE Aviation Systems (formerly Smiths Aerospace) [5] summarizes the shortcomings of federated architectures: *"Add to this picture the fact that some level of redundancy had to be built into each functional sub-system, and you're left with an overall system architecture that is inefficient, heavy, and expensive to develop and maintain. It's also void of much of the interaction (between sub-systems) that is now deemed essential in a true, modern system."*

In an *integrated architecture*, however, we treat function-specific applications and general-purpose platforms as two separate building blocks, instead of regarding each function as a monolith comprising software and computational hardware. RTCA DO-297 [7] defines an *application* as *"software and/or application-specific hardware with a defined set of interfaces that, when integrated with a platform, performs a function"*. On the other hand, *platforms* are defined as a combination of software and hardware to *"provide computational, communication, and interface capabilities for hosting at least one application. [...] Platforms by themselves do not provide any [...] functionality"*.

Unlike federated architectures, integrated architectures allow hosting several applications on one platform in order to share the platform's computational and communicational resources. This allows the system developer to reduce the number of platforms and therefore, reduce the system's weight, energy consumption, and costs. The weight and space savings of an integrated architecture in a passenger aircraft are, as an example, equivalent to at least two seats including passengers [8]. In addition to that, the sub-systems of an aircraft can more easily interact with each other to provide enhanced functionality.

From the safety perspective, the major downside of stronger integration is the loss of the natural fault containment barrier between separated platforms. In an integrated architecture, failures of one application may affect all applications hosted on the same platform, even if there are no functional dependencies between the applications. Therefore, the concept of integrated architectures is tightly coupled with the concept of *partitioning* [9]. A partition provides fault containment capabilities such that faults of an application in one partition cannot affect the platform's

capability to provide shared resources to applications in other partitions in such a way that the other applications fail.

However, in an integrated architecture that is not open, platforms are still developed for one specific system and application software is typically developed for one specific type of hardware, i.e., for one specific type of platform. In order to change this, the principle of *open architectures* is applied. In an open architecture, there are public standards that precisely specify the key interfaces between the modules of the system [10, 11]. In case of an integrated architecture, this key interface is the application programming interface (API) between application and platform. Consequently, we call an integrated architecture with a standardized API an *open integrated architecture*.

The API defines how the application interfaces with the platform in order to use the platform's shared resources. The common API of an open integrated architecture facilitates modularity and portability, since it allows the developers to reuse and replace platforms and applications and add new applications over the lifetime of the system. The API and the associated abstraction are provided by a collection of hardware-dependent software. Some authors call this software operating system (OS), whereas others use the term OS more restrictively to describe only a subset of all the functionality necessary to implement the API. For this reason, we refer to the software providing the API more generally as middleware or *platform software*. We define platform software as software that either directly (not via the API) accesses hardware or directly accesses other platform software modules. Figure 2 shows a platform software architecture example.

If an embedded system is designed as an integrated open architecture, the process for designing the system changes. Traditionally, embedded system design is a top-down process that starts with the specification of the system's functionality and continues with the specification of the system architecture until the process reaches the step where software and hardware are implemented in parallel. In an open integrated architecture, however, the standardized API of an execution platform allows decoupling the development of platforms from the function-specific development of applications. This changes the top-down process to a meet-in-the-middle process, as aptly described by [12]. In [12] this process is called *platform-based design* and we will introduce it in subsection 2.1.1.

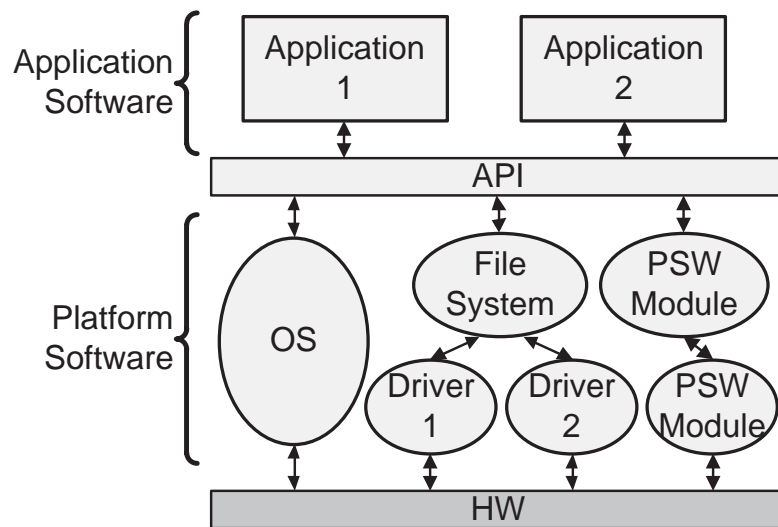


Figure 2: Simplified example a platform software (PSW) architecture in an open integrated architecture. Application software components are shown as rectangles, platform software modules are shown as ellipses.

The next subsections are going to introduce two very common open integrated architecture platforms. The **AUT**omotive **O**pen **S**oftware **AR**chitecture (AUTOSAR) from the automotive domain will be presented in subsection 2.1.2 and a civilian version of **I**ntegrated **M**odular **A**vionics (IMA) from the aviation domain will be presented in subsection 2.1.3. After that, we will present in subsection 2.1.4 a domain-independent description of platform services in open integrated architectures, which we will derive from the example platforms and use to define our specification language in the later chapters.

AUTOSAR and IMA are the two most important and wide-spread examples. There are, of course, other integrated architectures or operating systems with a standardized API in the embedded domain, such as VxWorks [13], QNX Neutrino RTOS [14], CodeSys RTE [15], Integrity [16], L4 micro kernels, PikeOS [17], or OSEK-OS [18]. For reasons of accessibility, we chose the two open and publicly available specifications mentioned above. However, we believe that the abstract platform services presented in 2.1.4 are able to cover most of the services provided by the other middleware products as well, especially since there are add-ons for some of the listed operating systems to make them IMA- or AUTOSAR-compatible.

2.1.1 Platform-based Design

The most typical embedded system development process is an adapted form of the traditional V-model. It is adapted in the sense that at a certain point in time, the developer has to decide which functionality to implement in hardware, which functionality to implement in software, and how to connect hardware and software with each other via

interfaces. From this point onwards, the system development splits up into hardware and software development running in parallel. In the development of open integrated systems, this process changes.

The standardized API of an open integrated architecture predefines the interface between application and platform as well as a large part of the functionality provided by the platform and the hardware, respectively. This fixed aspect in the design allows application and platform developers to implement their products largely independent from each other, and either development is possible without an embracing system development. The task of finally integrating applications and platforms into a system lies with the system integrator. First, the integrator defines the system's *functional architecture* by integrating all applications, and defines the system's *platform topology* by integrating the available platforms. In the next step, the integrator maps the applications onto the platforms during the so-called deployment phase yielding the integrated system. This platform-based design process is illustrated in Figure 3.

On closer inspection, deploying a functional architecture onto the platform topology is challenging for at least three reasons. First, the abstraction gap between a function and a platform is usually too big to directly map them to each other. Second, the deployment is constrained by the requirements of the application and the available capabilities of the platforms. And third, within the given constraints, it is still a challenging problem to find a suitable deployment. While the third point will be discussed in detail in section 2.2 of this chapter, we want to elaborate on the first two points in this subsection.

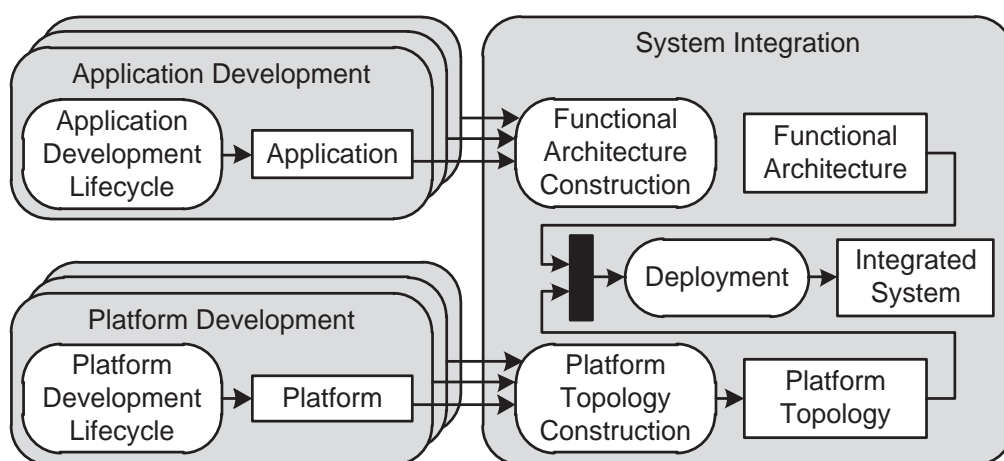


Figure 3: A development process for open integrated systems

If we regard an application as a stateful function, an application is simply defined by its inputs, outputs, and a corresponding transfer function. What we described earlier as functional architecture is created when the outputs of one application are connected with the corresponding inputs of other applications to model their interaction with each other. Such a

model does, for example, fall short of information about how and when to execute a function and how to implement communication needed to map the applications to the platforms. To bridge this gap, there has to be an intermediate model that can be directly mapped to the platform services. Such an intermediate model could, for example, map the functions and signals of the functional architecture to tasks and messages. These tasks and messages are then mapped to the execution platforms and communication channels of the platform topology as proposed in [19]. An example mapping is shown in Figure 5.

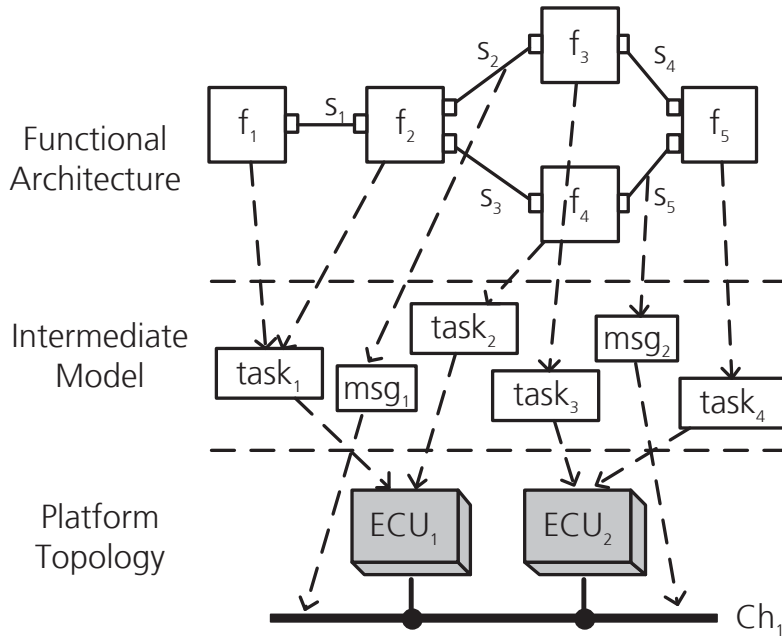


Figure 4: Mapping of a functional architecture onto a platform topology via the intermediate model (figure is based on figure 4 of [19])

Coming back to our second point, the necessity of checking whether a given platform meets the requirements of an application is an inherent property of platform-based design. Whereas in traditional development, the platform was tailored to the specific needs of the function, it is now a general-purpose component developed without knowledge of all the possible functions it may host. Therefore, the “*top-down constraint propagation and the bottom-up performance estimation*” [20] is a key aspect of platform-based design. The developer of an application specifies certain constraints regarding the behavior of the platform, whereas the platform developer has to specify the capabilities of the platform, as shown in Figure 5. During deployment, the fulfillment of the application’s constraints has to be checked against the capabilities of the platform³. While this process is necessary for all kinds of dependencies between platforms and applications, the specification of safety-related

³ The discussion of typical deployment constraints as well as deployment objective functions is part of the next chapter.

application requirements and the corresponding platform capabilities, as well as checking whether they match, is the key contribution of our work (see Contribution 1 and Contribution 2).

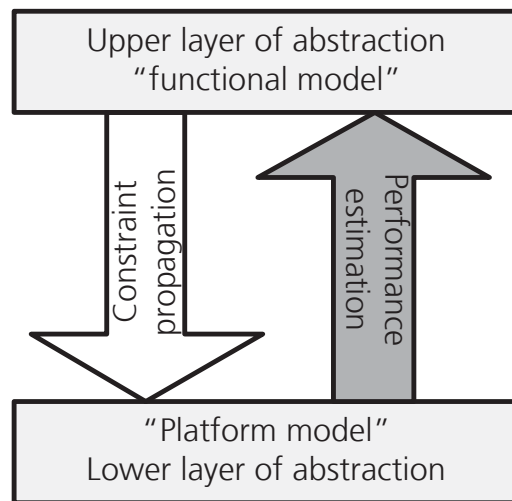


Figure 5: Interactions between abstraction layers in a platform-based design [21]

As a last point in this subsection, we want to mention that the idea of platform-based design can be used more generally than described so far. A platform can be more generally defined as an abstraction layer that hides the details of several possible implementation refinements of the underlying layer [21]. With this definition it is possible to define a whole platform stack looking at the development of a modern embedded system.

An API platform as specified by us provides an abstraction from the resources provided by the underlying hardware and hides its implementation details as well as the implementation details of the middleware. In addition to the API platform, there could also be a microarchitecture platform. In simplified terms, the microarchitecture platform provides a first layer of abstraction from the computation and communication hardware, which by itself is not abstract enough to be efficiently used by an application developer. Therefore, the API platform uses the microarchitecture abstraction layer to provide yet another, more convenient abstraction. Together, the API platform and the microarchitecture platform form the aforementioned platform stack. In this more general case, platform-based design is described as a meet-in-the-middle process, where successive refinements of specifications meet with abstractions of potential implementations.

We commonly find this kind of abstraction stack when we look at the implementation of an open integrated architecture. AUTOSAR, for example, defines a microcontroller abstraction layer that serves the sole purpose of adapting the higher-level middleware services to the specifics of the underlying microcontroller architecture. Many implementations of

the ARINC 653 API contain so-called microcontroller or microarchitecture support packages that can be exchanged together with the underlying hardware. The abstraction provided by such a microarchitecture layer, however, is beyond the scope of our work.

2.1.2 Example Platform 1: AUTOSAR

AUTOSAR [3] is short for AUTomotive Open System Architecture and is an open integrated architecture tailored to the needs of the automotive domain. Its development officially started in 2003, when a group of vehicle manufacturers and tier 1 suppliers signed the respective partnership agreement. The goal of the AUTOSAR development is to reduce hardware costs, manage the complexity of innovative functions, and improve the portability and reusability of applications and platforms [4]. Therefore, the development partnership defined a standardized middleware architecture with a standardized API as well as a development methodology for AUTOSAR-based applications. In this subsection, we first give an overview of the development methodology before introducing the AUTOSAR middleware architecture. Since the AUTOSAR standard is still evolving, we need to mention that this subsection addresses the third revision of the fourth version of the AUTOSAR specification, updated in January 2012.

Comparable to the development process described in subsection 2.1.1, the main concern of the AUTOSAR development process is the separation of application and platform development. To this end, AUTOSAR uses a concept called the Virtual Functional Bus (VFB). To use the VFB, application developers model their applications, structuring them into application software components and defining the components' incoming and outgoing signals, as well as the middleware services directly used by the components. The VFB then provides a virtual communication channel allowing the developers to logically connect application software components and to describe their interplay without anticipating their deployment and the resulting physical implementation of their communication. When finally deploying the application software components to specific platforms, the integrator has to choose a suitable implementation for the virtual bus, using available and applicable communication mechanisms. One can regard the VFB as a mechanism for defining a functional architecture.

In general, an AUTOSAR platform provides its services to applications via standardized platform software components, called basic software modules in the AUTOSAR context. Since application software components and the basic software modules are compiled independently, they have to be linked together via a third abstraction layer, called the Runtime Environment (RTE). Typically, the integrator generates the RTE automatically, using the information from the model-

based representation of the applications and the platforms. As of late, the RTE has been implementing more and more functionality that can be generated conveniently if the deployment is known. The AUTOSAR development approach is illustrated in Figure 6.

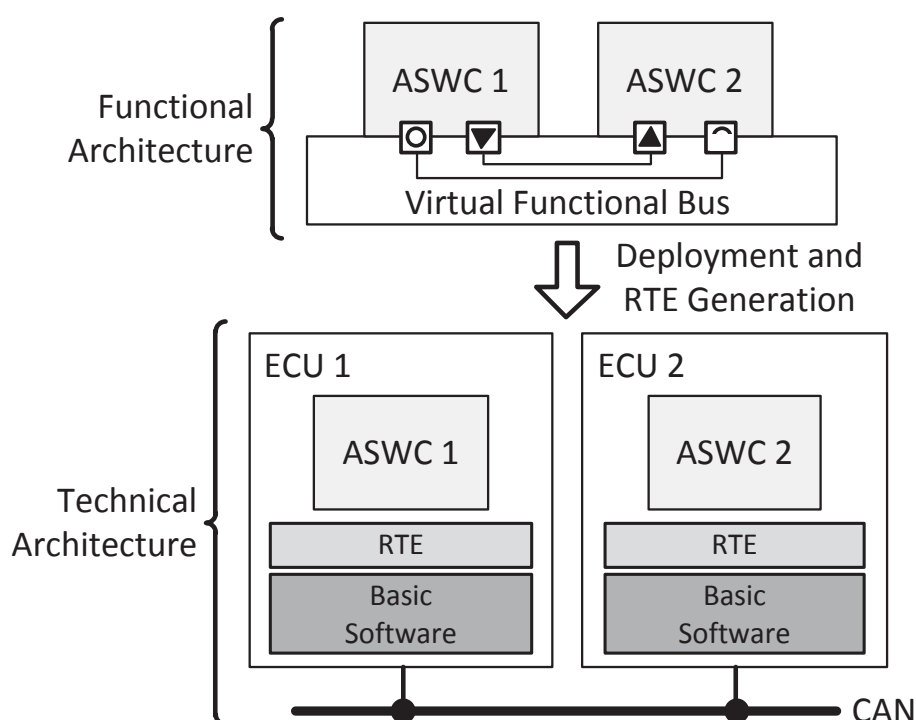


Figure 6: AUTOSAR methodology excerpt : From VFB to configured RTE [22]

In the following paragraphs, we will describe the AUTOSAR platform software. Even though we have learned that applications directly interface with the RTE, the RTE is ill suited to analyzing the services of the platform software. This is because the actual interface of the RTE is generated dynamically during system configuration and the generic RTE interface hides the functionality provided by the platform. In order to identify the services provided by the AUTOSAR platform, we therefore have to analyze the top-level interface of the basic software instead.

The basic AUTOSAR software consists of 66 individual modules, and there is a detailed interface specification of each module. However, not all interfaces are relevant to us, since most of them are not visible at the application layer. In fact, the basic software is again divided into three layers as shown in Figure 7. The lowest layer is called the microcontroller abstraction layer. It mainly consists of device drivers, which provide an abstraction from the differences of the underlying microcontroller hardware and the ICs connected to the controller. The second layer is called hardware abstraction layer and additionally abstracts from the specific layout of the microcontroller board. It abstracts, for example, from a CAN (Controller Area Network) channel that is implemented directly by the μ Controller or a channel that is implemented by a

separate CAN driver IC that is connected to the μ Controller via SPI. The third layer is called the service layer. This layer finally provides a convenient set of system functions to access the platform resources. This service layer is directly accessible for applications (via the RTE “glue code”).

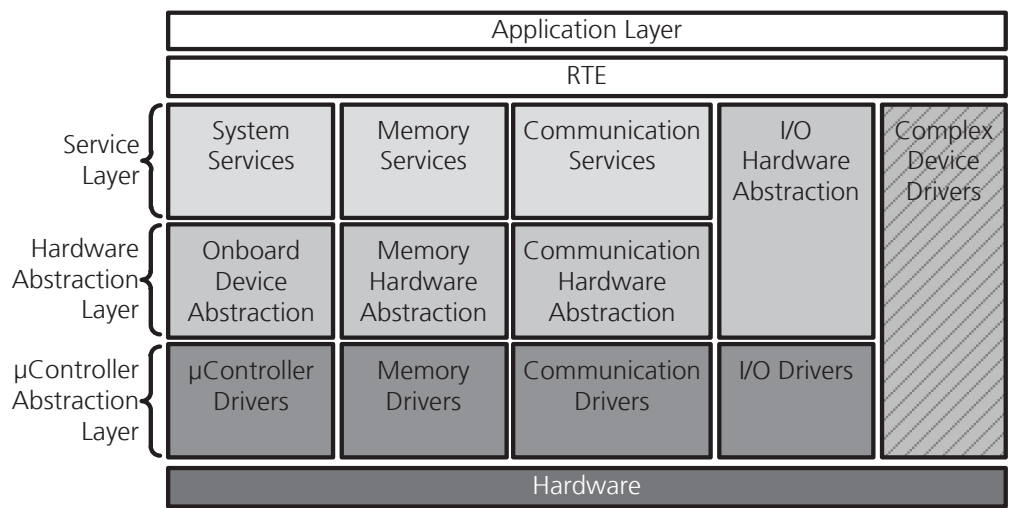


Figure 7: The layered platform software architecture of AUTOSAR [23]. The layers are differentiated by different shades of gray, from lighter to darker: the service layer, the hardware abstraction layer, and the microcontroller abstraction layer. Complex drivers allow for the integration into the platform software of software components that are not standardized by AUTOSAR.

The horizontally layered architecture of the basic AUTOSAR software is again sectioned vertically. Each of the four resulting vertical “stacks” represents one key service class of an AUTOSAR platform. These four service classes are:

System Services are used to manage, configure, and retrieve the status of the platform, as well as to perform inter-partition communication, diagnostics, and health management. The operating system is also part of this service class.

Memory Services allow access to non-volatile memory (NV-RAM). This class abstracts from the differences between EEPROM and Flash, and allows accessing external memory ICs via SPI. It furthermore provides convenience functions for storing redundant memory blocks and for memory consistency checks.

Communication Services implement services to send and receive messages to and from inter-platform communication busses. The layered communication stack abstracts from different bus types like CAN, TT-CAN, or Flex Ray and allows for large or multiplexed messages.

I/O Hardware Abstraction⁴ provides access to input and output devices to read from sensors and to control effectors. There is support for digital and analog I/O as well as for PWM signals.

Enabling safety is one of the primary objectives of the AUTOSAR development [24]. Therefore, the AUTOSAR standard contains a number of safety mechanisms ranging from end-to-end communication protection, logical program flow monitoring, or timing and memory protection. A summary of the safety-related requirements and features of an AUTOSAR platform is specified in the Technical Safety Concept Status Report [25].

Consequently, one goal of our technique was to be able to cover the AUTOSAR-specific safety requirements and to be able to automate the process for checking and arguing the sufficiency of these requirements in the face of a concrete application. In chapter 7, we will show how the AUTOSAR safety mechanisms can be modeled using our approach.

2.1.3 Example Platform 2: IMA – ARINC 653

IMA [26] is short for Integrated Modular Avionics and represents the overall movement of using integrated architectures in the avionics domain. There is no such thing as one central IMA standard; rather, there are several specifications that provide a standardized API that follows the concepts of IMA. There is, for example, the ARINC 653 [2] standard in civil aviation, the Def. Stan. 00-74[27] standard in the military domain, or Honeywell’s DEOS (Digital Engine Operating System) [28].

In this section, we will focus on describing the publicly available ARINC 653 specification, more specifically the first part of the standard, which deals with the core services of the platform software⁵. The ARINC 653 specification calls its API the Application/EXecutive (APEX) interface. The APEX interface integrates the application software with the platform software, called O/S kernel in the context of ARINC 653.

The application software is bundled into different application partitions by the application developer. The software in each application partition may only use the APEX interface, which makes this software platform-independent. Additionally, the application developer may provide so-called system partitions. The software in these partitions is allowed to additionally call non-standardized system-specific middleware functions. The software in application partitions as well as in system partitions is protected from mutual interferences and runs in user mode. However,

⁴ AUTOSAR does not offer an I/O Service layer.

⁵ The second part contains an extension of the API for file and database handling.

since the software in the system partition does not exclusively call APEX procedures, this software is not necessarily portable to other platforms, since these platforms might not support these system-specific functions. Figure 8 shows the relationships between the different types of software in an ARINC 653 system.

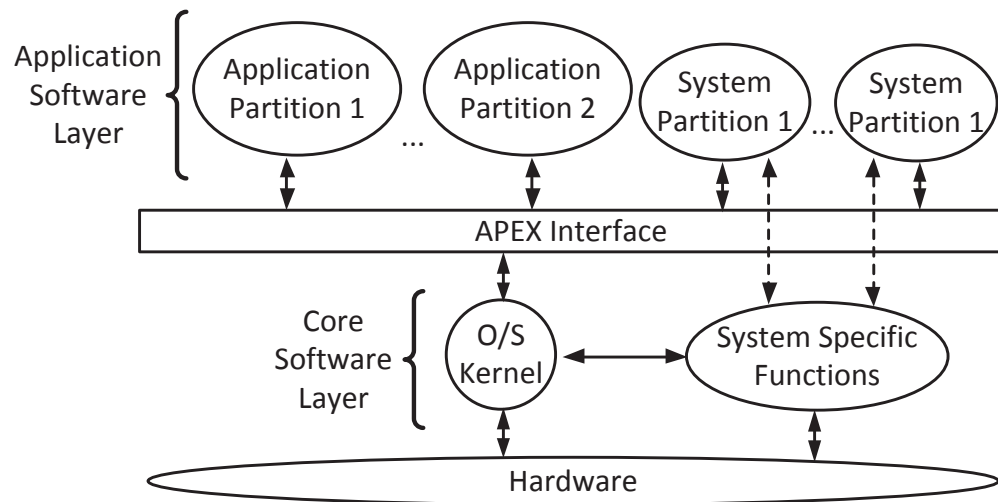


Figure 8: ARINC 653 software architecture and its relations [2]

The handling of communication between applications is comparable to the AUTOSAR approach. The application developer does not know whether communicating applications are on the same platform or not. Therefore, the integrator has to configure the APEX appropriately after the deployment is fixed. However, the handling of communication is different from AUTOSAR in at least one aspect. The developer of an application software component in an ARINC 653 scenario does know whether the communication counterparts are in the same partition. Therefore, there are different service calls for inter-partition and intra-partition communication in ARINC 653, whereas there is only one communication interface visible to AUTOSAR applications. The differentiation between inter- and intra-partition communication is made after the application interface in an AUTOSAR system

Just like we listed the AUTOSAR platform services by analyzing the API, we extracted the following ARINC 653 service classes from analyzing the APEX interface:

Partition management is used to retrieve and control the status of partitions. A partition provides an area of fault containment as specified in section 2.1.

Process management is used to create, stop, and restart processes as well as to retrieve and control their status. A process in ARINC 653 has its own memory area and may possess platform resources.

Time management is used to retrieve time information or wait until a certain time interval. This service class includes functions for retrieving a global time

Inter-partition communication is used for communication between partitions on the same as well as on different platforms.

Intra-partition communication is used for communication and synchronization between processes in the same partition (e.g., semaphores, events, and buffers)

Health monitoring is used for handling detected failures. This includes default reactions like shutting down partitions or processes, as well as invoking application call-back functions.

ARINC 653 addresses safety explicitly through the specification of robust partitioning mechanisms as well as with its health monitoring services. Just like the mechanisms introduced by the AUTOSAR standard, these features must be describable with our specification language as we will show in chapter 7. Additionally, there is a standard for certification in the IMA domain, RTCA/DO-297 [7], which will be introduced in subsection 2.3.1.

2.1.4 Platform Service Types

While there is a wide variety of platforms, most of the platforms provide comparable services. Although specific services might be available (or not) for specific platforms, and although the same service might differ in aspects like performance or call syntax, the core of the provided services is relatively stable. This circumstance is also the reason why industry was able to standardize platforms in the first place. This subsection provides a list of abstract platform services derived from the AUTOSAR and the ARINC 653 examples. In this section, we introduce this list of standard services that a platform typically provides. We derived this list from standard platform specifications like AUTOSAR (subsection 2.1.2) or ARINC 653 (subsection 2.1.3).

With the exception of the computational capabilities provided by the CPU and the storage capabilities provided by the main memory, applications access a platform's services via the API provided by the platform software. With regard to platform software or operating systems, Tanenbaum [29] differentiates between two main tasks:

1. **The platform software as an extended machine**
2. **The platform software as a resource manager**

The first task regards the platform software as an extension of the machine that is the underlying hardware. The hardware itself already comes with an interface that allows the software to interact with the CPU and its devices, but there are two main reasons why this interface is extended by the platform software. The first reason is that the direct hardware interface is usually complicated to use, which is why the platform software provides an easier and more convenient interface for accessing the platform's resources than the direct HW interface does. Second, the platform software abstracts from particularities of different hardware implementations and therefore provides a hardware-independent interface. The application software directly calls this abstract/extended interface and does not see the actual hardware anymore. This is why Tanenbaum refers to this as the top-down view on platform software.

The second task becomes relevant if there are multiple applications sharing the platform's resources. In this case, the platform has to manage and regulate the application's resource usage, such that the applications are able to share the common resources properly, which enables what is usually called multiprogramming. In the context of safety-critical systems, platform software must provide additional and stricter guarantees with regard to the absence of interferences via shared resources. We have referred to mechanisms providing freedom from interference as partitioning earlier in our work (section 2.1). In contrast to the services of the platform as an extended machine, the services that manage and regulate concurrent resource usage are usually transparent to applications. In fact, there is typically no direct interaction between applications and these services, meaning that there is no explicit API to influence these services. Therefore, Tanenbaum refers to this as the bottom-up view on platform software.

This second task can be further differentiated according to the resource that is shared and protected. There are those resources that are accessed via the API, such as communication channels, I/O devices or files, and there are resources like RAM and CPU, which are typically not accessed via the platform software. In the latter case, the protection and management of the resource has to be jointly implemented by platform hardware and platform software since otherwise, the platform software would have no means for controlling access.

In order to provide a more detailed view on a platform's typical API, we used the specification of AUTOSAR and ARINC 653, as well as the structure proposed by [29] to split the services into eight different service types. A mapping of the AUTOSAR and the ARINC 653 specifications to these service classes can be found in section 7.1.

(1) Synchronization mechanisms providing measures for synchronization between tasks and for the implementation of

critical regions: In the literature, the term *inter-process communication* (IPC) is typically used as a collective term for mechanisms enabling communication between processes running in different memory areas. This includes services for synchronization, such as barriers or events, services for realizing critical regions, such as spin-locks or binary semaphores, and mechanisms for exchanging data, such as buffers. Since we cannot assume that there is a memory management unit (MMU) in every embedded platform, we do not want to use the pre-allocated term IPC. Instead we use the term *synchronization mechanisms* for our first service class, which includes mechanisms for synchronization and for the implementation of critical regions. As we do not assume that every μ Controller supports the concept of processes, we do not differentiate between inter-process and intra-process communication as often done in literature. Communication in general is captured by the next service class

(2) Communication for information exchange between tasks: This service class contains all mechanisms that allow applications to exchange data. Since we want to describe services only from the application's point of view, we do not differentiate between inter- and intra-process communication, between inter- and intra-partition communication, and between inter- and intra-platform communication. This kind of differentiation is only made during deployment and is, therefore, transparent for the application.

(3) I/O access for reading and writing from/to sensors/actuators: This class is comprised of services to read from input devices such as A/D converters or digital input channels and to write to output devices such as PWM channels or digital output channels. We do not differentiate between I/O channels that are on the chip of the μ Controller (such as an internal A/D converter) and external channels that are implemented by external ICs attached via local busses, such as I²C or SPI.

(4) Time services for measuring and waiting a certain time: Time services contain services for measuring relative time, for waiting a certain time, as well as for retrieving a global time. Relative time is characterized as a time interval between two events, e.g. between a start and a stop timer call. Global time, on the other hand, provides a consistent and comparable time base for the overall system.

(5) Memory services for accessing memory that is not directly mapped onto the address space or for addressing mapped memory more conveniently: This class aggregates services for indirectly reading and writing to non-volatile memory. Unlike direct memory access, indirect memory access is performed via the platform software API and not directly on the memory bus. This can include simple convenience functions for writing to or reading from Flash,

EEPROM, FeRAM, or a hard disk. This can also include more sophisticated functions for storing and retrieving data in databases or via a file-based service.

(6) Health monitoring for detecting and handling application and platform failures: This is a category for all services that address detection of application as well as platform failures and failure handling in general. Typical failure detection mechanisms include deadline monitoring or logical sequence monitoring. Typical failure reactions include restart or shutdown of partitions or the platform, as well as setting default outputs or sending default messages. Besides that, health monitoring contains services for self-testing, such as built-in self-tests of the hardware like memory or computation logic checks.

(7) Basic computation: This category summarizes all services of the platform that are not accessed via the platform software API. Typically, these are the computation services of the CPU and the data storage services of the main memory. Please note that the absence of an API does not imply that the access to these resources is not coordinated by the platform software.

These seven service classes identified by an analysis of the two most common open integrated architectures are the foundation for the definition of the specification language provided by this thesis (see Contrib. 1). Please note that this implies that most open integrated architectures are statically configured. It is, for example, not possible to allocate memory or to create operating system objects dynamically during runtime.

2.2 Deployment Evaluation

In the previous section, we presented the typical development lifecycle for open integrated systems, which included a work step called deployment. This section presents the topic of deployment evaluation. Deployment evaluation has many different evaluation criteria, which is why we will introduce every safety-related criterion in the following subsection. Together with these criteria, we will present the corresponding related approaches and name the criteria that our approach evaluates and what is unique about what we do.

The term deployment describes the process of mapping a system's functional/logical architecture onto the system's technical/physical architecture. In this context, the latter is sometimes also called the target environment of the deployment. A functional architecture consists of functions interconnected by signals, whereas the physical architecture consists of computational nodes interconnected by communication channels and gateways that interconnect the communication channels.

Functions are implemented by software components⁶ that are mapped to computational nodes, and signals are packed into messages that are mapped to communication channels. Since functions are always implemented by applications (see definition of application), we usually refer to the software components implementing a function as application software component (ASWC).

The idea of mapping ASWCs to platforms can be regarded as the top-down view on deployment. From the bottom-up perspective, however, the platform typically does not perceive the application as a collection of components. Instead, the platform perceives applications as a collection of schedulable entities, which we refer to as tasks. Therefore, in the literature one may also find the term task allocation alternating with the term deployment, depending on the author's viewing angle.

In the previous subsection, we argued that there must be a standardized interface between the functional and the physical architecture to enable separate development of applications and platforms while maintaining deployment compatibility. In the context of distributed embedded systems, open integrated architectures contain such standardizations. In the object-oriented domain, however, such standards have long been made available by middleware standards like the Common Object Rquest Broker Architecture (CORBA) [30] or comparable methods. A well-known source for information about object-oriented deployment is provided, for example, by the “*Deployment and Configuration of Component-based Distributed Applications Specification*” [31] published by the Object Management Group (OMG).

According to [31], a deployment process can be separated into five steps. Even though our approach only focuses on the third work step, we will, for reasons of demarcation, introduce the other four as well. **(1)** The *installation* step is performed by the developer of an application and describes the act of bringing an application's software components into a software repository. This does not include actually moving the software to the target environment, but is a preparation step that enables the second work step. Since we think that the installation step is not of equal importance for embedded systems, we will not further elaborate on it. **(2)** The *configuration* work step allows the developer to configure the application in the repository, for example changing the acceleration ramp of a cruise control application. In the embedded domain, this is typically done by the application developer. Since we intend to define deployment as a process that is performed by the system integrator, we do not include configuration in our evaluation. **(3)** In the *deployment planning* phase, the integrator plans the mapping of the system's

⁶ Functions can also be implemented by hardware components. Hardware components are, however, extraneous to deployment.

functional architecture to the system's technical architecture, for example the mapping of the software components to execution platforms or the mapping of the logical signals to communication links. The decisions made during the planning phase are specified in the *deployment plan*. Deployment planning does not include actually moving the compiled software to an execution node. **(4)** *Preparation* is the work step in which the integrator configures the target environment such that the planned deployment can be executed. We will refer to this step as *configuration*, since in our experience this is more typical in the embedded domain. **(5)** In the *launch* step, the application is finally moved to its target and executed.

During deployment planning, there typically is a solution space containing numerous possible solution candidates for mapping a given functional architecture onto a given technical architecture. *Deployment evaluation* is our term for the qualitative or quantitative assessment of *deployment solution candidates* aimed at exploring the design space and find the most suitable deployment plan.

Deployment has several different quality goals that can be evaluated. The evaluation can be performed in a qualitative pass/fail manner using constraints, or in a quantitative manner using *objective functions*. Objective functions can again be divided into *fitness functions*, if the assessed criterion has a positive nature and is to be maximized and *cost functions*, if the regarded criterion is negative and is to be minimized. Finally, an objective function can also be used to implement a constraint if the user defines a pass/fail criterion using a minimum or a maximum threshold, respectively, for the objective function.

Example: *The workload of the platform must be lower than 67%.*

In the following subsections, we will classify evaluation approaches with regard to the criteria they evaluate. Table 1 shows an extended version of a table taken from [32] listing the deployment criteria we used for the classification of the related work. As the objective functions introduced by this thesis perform a safety-related deployment evaluation, the related work lists only safety-related approaches and the listed criteria have a safety focus as well. Consequently, the evaluation criteria list is not complete since there are other important, not directly safety-related criteria, such as the exploitation of computational concurrency.

The following subsections will introduce the related work, classified according to the evaluation criteria they address.

Table 2 provides an overview of the different deployment qualities addressed by each approach we have analyzed.

Table 1: Quality criteria of a given deployment solution.

Criterion	Description	Desired
Match	Analyzing how well application requirements match platform capabilities (e.g., schedulability)	Maximize
Delay	Analyzing the end-to-end delay of an application	Minimize
Flow	Analyzing messaging traffic and network capabilities	Minimize
Replication	Analyzing the costs for replicating components	Minimize
Reliability	Analyzing the reliability or failure rate of the application	Maximize
Mixed criticality	Analyzing the criticality heterogeneity of software components in one partition	Minimize
Fixed assignment	Explicitly mapping a software component to a specific node	Constraint
Diverse assignment	Forbidding to map two or more software components to the same node (or node type)	Constraint

Table 2: A mapping between quality criteria and references. An “x” indicates that the approach denoted by the column evaluates the criteria denoted by the row. An “x” in parentheses indicates that the criteria are evaluated but without safety focus.

Quality \ Reference	A	B	C	D	E	F	G	Our approach
Match		(x)	(x)	(x)	(x)	(x)		x
Delay	x	x						
Flow		x	x	x		x		x
Replication	x	x		x			x	
Reliability	x	x					x	
Mixed criticality								x
Fixed assignment	x		x			x		x
Diverse assignment			x		x	x		x

Table 3: A mapping of identifiers used in Table 2 to references.

<i>ID</i>	<i>Name</i>
A	Fault-Tolerant Distributed Deployment of Embedded Control Software [33]
B	Automated Deployment of Distributed Software Components with Fault Tolerance Guarantees [34]
C	Two Optimization Techniques for Component-Based Systems Deployment [35]
D	Effective distribution of object-oriented applications [32]
E	Task allocation in fault-tolerant distributed systems [36]
F	Allocating hard real-time tasks: an NP-hard problem made easy [37]
G	Task allocation algorithms for maximizing reliability of distributed computing systems [38] <i>and</i> Models and algorithms for reliability-oriented task-allocation in redundant distributed-computer systems [39] <i>and</i> Safety and reliability driven task allocation in distributed systems [40]

2.2.1 Match

The *match criterion* measures how well the capabilities of a platform match with the requirements of an application⁷. A very typical manifestation of this quality is measuring how well the computational power of a platform's processor(s) matches with the required computational power of an application. Maximizing this particular aspect equals maximizing the workload of processors, which again may result in a lower number of required platforms. There are also approaches working in the opposite direction, like equally balancing the workload of all platforms as proposed by [36]. According to [36], this may lead to more spare time for performing diagnosis functions and, therefore, increasing reliability. These two views on processor work load can be applied more generally to the match quality. The overfulfillment of requirements typically leads to an inefficient use of available resources, which is expensive. The plain underfulfillment of requirements leads to failures, and a very close fulfillment of requirements might lack the safety margin necessary for resilience against adverse circumstances.

So in order to check whether the platform fulfills the minimum requirements of an application, the match quality is often used as a

⁷ The corresponding meet-in-the-middle development process is described in subsection 2.1.1

constraint. Typical requirements ask for the availability of a floating-point-unit (FPU), a certain amount of available primary and secondary storage space, minimum computational power, support for concurrency, or supported security levels as listed in [32]. Especially checking whether the current deployment does not exceed the available memory capacity [36, 37] or the available computational power [34] is very common. Other approaches go one step further than just checking required against available computational power. The techniques presented in [36] and [37] try to automatically compute a schedule in order to check whether the current deployment solution is valid with regard to the available computational power and the deadlines of the applications.

The method defined in [35] allows the designer to specify classes of nodes (like x86/Windows computer) and classes of software components (like Windows application) and to define “supports” relations between the classes. The deployment algorithm then checks whether the class of each node instance supports the class of each hosted ASWC instance. This technique allows for the specification of high-level match constraints.

Regarding the example above, the match quality contains a lot of aspects that are not directly safety-relevant. On the other hand, there are numerous safety requirements that an application may demand from its host platform. These include fault-tolerant communication, robust partitioning, the detection of missed deadlines, and many more. Our approach allows specifying the safety-related requirements and capabilities of applications and platforms on a detailed level (see **Contrib. 1**), as well as checking whether they match (see **Contrib. 2**), which is unique in the field of deployment evaluation to the best of our knowledge.

2.2.2 Replication and Reliability

Redundancy is a very common architectural pattern for safety-critical systems. Critical functions are implemented and executed redundantly, and an arbitration component chooses which of the redundantly computed results to use, or how to integrate multiple valid results. In case of a system that always fails silently, dual redundancy is sufficient to tolerate single failures. If the system does not only fail silently, the system requires triple redundancy to tolerate a single failure (byzantine failures might even require a higher level of redundancy). A simple replication of a software application is, however, not suitable for dealing with systematic failures. These failures will most likely occur on all instances of the redundant software simultaneously, rendering redundancy useless. Nevertheless, random failures of the hardware platform can be tolerated if using replicated application software components that are deployed to different platforms. On the other hand, replication of software

components increases the required communicational and computational capacity and can, therefore, have significant effects on a system's hardware costs. Consequently, it is important to make efficient use of replication during deployment to find a suitable reliability/cost tradeoff, which is modeled by the *replication criterion*.

Besides fault tolerance strategies, the designer can also choose to use highly reliable components in order to decrease the failure rate of a system to an acceptable level. If there are platforms and communication channels that have relatively high and low failure rates, the deployment has an effect on the overall reliability of the system. The effect of deployment on quantitatively measured reliability is described by the *reliability criterion*.

The approach described in [33] optimizes the reliability/cost tradeoff by calculating the optimal number of software component replicas and by deploying them efficiently to a given platform topology. To do so, the designer has to specify the desired fault behavior of the application together with the application's fault scenarios. The desired fault behavior is defined as a minimum set of functions that must remain available under certain fault scenarios. Every fault scenario is defined as a set of platform and network failures that must be tolerated by the application simultaneously. Using these inputs, an optimization algorithm calculates the number of required replicas, as well as a deployment plan to ensure the desired fault behavior in the presence of the specified fault scenarios.

The method proposed in [34], on the other hand, minimizes the failure rate resulting from a deployment by taking the failure rates of the different platforms and communication channels into consideration. The objective function estimates the failure rate of the system by summing up the specific failure rates of the nodes and communication channels needed to compute the application. If an application software component is duplicated and the replicas are deployed to different platforms, the corresponding failure rate is assumed to be zero, since the fault model only considers single failures. The same is true for the duplication of communication links. By summing up the failure rates in a series connection, the algorithm approximates the overall failure rate. Using the logical "or" would, however, yield the exact failure rate.

Building further on [39], a more precise failure rate calculation is provided by [41]. The algorithm assumes that all hardware components, computational nodes, and communication links have different but constant failure rates. To calculate the mission failure rate, the algorithm integrates the failure rate over the accumulated execution times of all software components and the accumulated transmission times to perform all necessary communication over the mission time of the system. The reliability formula also accounts for redundancy in the application model. The method proposed by [40] also builds on the

reliability function provided by [39] and adds a feature for modeling fail-safe mechanisms. Instead of calculating only the system's reliability, the algorithm accounts for the probability of detecting failures and transitioning into a safe-state, for example, by shutting the system down.

2.2.3 Delay and Flow

Timing behavior is an important aspect of many embedded systems. Many applications in the embedded domain are implemented as closed-loop control. Such an application perceives its environment using sensors, then uses this information to compute a control value, and finally influences its environment via effectors such that a certain set value is reached. Unexpectedly high delays or varying delays (jitters) are not accounted for in the underlying control theory and are, therefore, detrimental to the accuracy and stability of the control loop. Besides closed-loop controls, safety mechanisms often have to react to failures within a short time so that the fault tolerance time of the respective system is not exceeded. The deployment has a notable effect on the end-to-end delay and the jitter of an application. This is due to delays introduced by software components that are executed asynchronously on different nodes and the resulting wait times, as well as the delay introduced by signal transmission. The *delay criterion* measures the effect of the deployment on the jitter and the end-to-end delay of applications.

Besides the resulting delay, communication is also restricted by the bandwidth of communication links. In many modern cars, for example, communication networks operate at their limit, and adding new communication busses is avoided because of weight and costs. When the communication link uses priority-based access arbitration, delay and jitter are also affected if the workload of the bus increases. Quality aspects regarding the total number of exchanged messages compared to the available bandwidth of the communication channels are summarized by the *flow criterion*. The most intuitive way to approach this aspect is to compare the available bandwidth of the communication channels with the required bandwidth resulting from the deployment as done in [37] and [34].

The approach described in [33] allows a control theorist to specify the maximum time (T_{max}) that is allowed to elapse between reading the sensor values and controlling the effectors of a closed-loop control, which is what we called end-to-end delay above. Knowing the required and available computational power of each software component and execution node, the proposed algorithm calculates a deployment plan and a corresponding schedule, such that T_{max} is not exceeded. To account for communication delay, the algorithm uses the worst-case

communication delay specified a priori for each communication link by the developer.

Instead of using delay as a constraint, the method proposed by [34] minimizes the end-to-end delay in a best effort way. Besides that, the computation of the actual delay is comparable to the computation done by [33] described above. Considering that a control loop should typically not be computed as fast as possible but with the exact delay specified by the control theorist, using delay as a constraint seems more favorable for embedded systems than optimizing delay.

The modeling language specified in [35], on the other hand, allows specifying the frequency of the messages exchanged by the software components. The objective function then multiplies the frequency of each message with the number of *hops* required to transmit the message in order to calculate and optimize the number of messages sent per second. A transmission requires several hops if a gateway is used to route the message over multiple communication channels.

The approach presented in [32] also evaluates message traffic and thus the flow criterion, but is based on the analysis of scenarios. Each scenario is evaluated individually regarding the required communication and the likelihood of the scenario. In a second step, the scenario specific data are combined with the likelihood of each scenario in order to get an estimation of the traffic encompassing all scenarios. Based on this information, the communication is optimized on two levels, inter-site and intra-site communication. Intra-site communication describes message exchange between two computers that are located close to each other and that are connected via a local communication channel. Such a channel allows for cheap communication regarding the available bandwidth and the transmission delay. Inter-site communication, on the other hand, is expensive and must therefore be reduced with higher priority.

The flow criterion is also regarded by our deployment evaluation algorithm (see Contrib. 3). Instead of measuring the required bandwidth or the number of messages per second, we evaluate the costs caused by the safety mechanisms needed to protect against communication failures. The algorithm differentiates between two levels of communication as well, comparable to the inter-site/intra-site differentiation.

2.2.4 Fixed and Diverse Assignment

Even though we assume that deployment is flexible, i.e., each platform is, in principle, compatible with each application software component, there are some reasons for restricting the deployment solution space.

We differentiate between two kinds of restrictions, *fixed assignment* and *diverse assignment*. Fixed assignment allows the integrator to directly assign a software component to a specific platform. This is necessary, for example, to put the software component performing input conditioning of a sensor value onto the same platform the sensor is attached to. Fixed assignments are supported by [33], [37] and [35].

Diverse assignment or *separated assignment* [35–37], on the other hand, allows the integrator to forbid putting a set of software components onto the same platform. This is typically done to increase reliability by forcing a mapping of software components replicas onto different platforms.

Besides fixed and diverse assignment, [35] allow specifying the constraint that two components must be placed on the same platform. This is mainly done to avoid network communication between tightly coupled applications. If communication traffic is already measured using the flow criterion, this constraint might not be necessary.

Our evaluation approach (see Contrib. 3) allows fixed as well as diverse assignment. In the case of diverse assignment, we differentiate between diverse assignment for protecting against random failures and diverse assignment for protecting against systematic failures. To protect against systematic failures, application software components must not only be deployed onto different instances of the same platforms, but the platforms must be of different kinds.

2.2.5 Mixed Criticality

If there is the possibility that a set of software components interferes with each other, safety standards typically demand that all software components in the set are developed according to the highest integrity level of all software components in the set. This is done to avoid that failures of lower-criticality components, developed according to less strict development requirements, cause higher-criticality applications to fail and therefore, indirectly cause hazards with higher criticality.

We introduced the concept of partitioning in section 2.1. Partitioning guarantees freedom from interference between software components located in different partitions. If, however, software components with mixed criticality are in the same partition, the aforementioned rule applies and the criticality level of the applications is increased. This also increases development costs and, therefore, should be avoided if possible. These costs are evaluated using the *mixed criticality criterion* and are assessed using our deployment evaluation technique (see Contrib. 3). To the best of our knowledge, there is no other method that allows optimizing this criterion.

2.3 Modular Certification

In the previous section, we discussed several criteria for evaluating a deployment. The match criterion is one of them and is used to evaluate how well the requirements of the application match the capabilities of the platform. This process of matching application requirements with platform capabilities, however, anticipates that the safety-related dependencies between application and platform are specified modularly. Traditionally, though, safety-critical systems are certified in their entirety. This means that even though the system might be composed of individual parts, the safety of the composite is argued monolithically. Since certification is a significant matter of expense in the development of a safety-critical system, the reuse of components is only efficient if a large portion of their certification artifacts is reusable, too. Furthermore, the deployment cannot be handled flexibly if the safety-related dependencies between applications and platforms are not specified modularly and, therefore, cannot be checked after deployment. Along these lines, Rushby defines the concept of *modular certification* as “the development of modular components that could be largely “precertified” and used in several different contexts within a single system, or across many different systems”⁸.

Modular certification is a wide field of interest. In the following section, we will describe a classification containing the different aspects of modular certification. We will use this classification to emphasize which aspects of modular certification our work addresses. Furthermore, we will use this classification to categorize the related approaches in order to show how our work relates to existing approaches, and to isolate that gap in the research landscape that we meant to fill with this thesis. The classification contains three major branches: (1) **The interface constituents** describe the different kinds of information that have to be contained in the public certification artifacts of a product to make the product reusable efficiently. We will call this public part of the certification the interface of the modular certificate. (2) **The interface orientation** describing the different kinds of interfaces a modular certificate may have, especially in the context of open integrated architectures. (3) **The work steps** involved in using a modular certification. The classification is illustrated in Figure 9.

The typical way for specifying a component that is interconnected with other entities in a self-contained, i.e., modular, manner is to capture its dependencies to other entities using an interface specification. The first aspect in our classification differentiates the different interface

⁸ The definition of modular certification has been slightly broadened. The term airplane has been replaced with system, since the author was only referring to the certification of airplanes in his original text.

constituents that must be captured in a modular certificate's interface. To identify the information that has to be captured in such an interface, we examine the definition of fail-silent taken from the aviation standard [42]. In the standard we find that certification requires (A) an assessment of whether the design of the product is applicable to demonstrate an acceptable level of safety, and (B) a judgment that confirms that the product conforms to the afore-assessed design. In other words: (A) checking whether the system is safe as specified and (B) checking whether the system has actually been developed as specified, i.e., whether the system meets its specification.

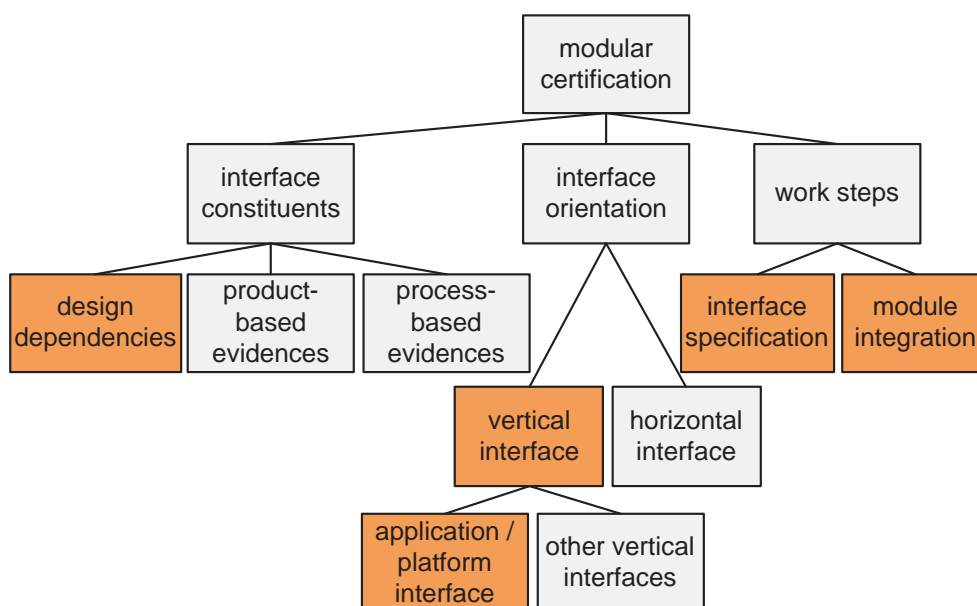


Figure 9: A classification of the different aspects of modular certification. Aspects that are covered by this thesis are depicted by dark gray boxes.

The first aspect (A) is analyzed and argued on the basis of design documents such as requirements or architecture specifications that describe how the system is structured and how it behaves. In order to modularize such an argument, one must make assumptions about the behavior of related components and give guarantees regarding its behavior, which is what we summarized as design dependencies in our classification. The latter aspect (B) is argued on the basis of so-called evidences, typically in the form of development records [43], [44], and [45] differentiate between *product-based evidence* and *process-based evidence*. Product-based evidence, such as testing, verification, or review records, directly demonstrates the compliance of a product with its *design specification*. Process-based evidence, such as lifecycle data or staff training records, demonstrates compliance with a given development process. The motivation for showing compliance with the development process is the establishment of the validity and trustworthiness of product-related evidence and therefore, indirect support for the claim that a product meets its design. There are some

who believe that mature and strict processes have a direct effect on the safety of a system, but there have been no empirical data to underpin that claim. The dependencies between product-related evidences, process-related evidences, and design specification are shown in Figure 10.

Consequently, the interface of a modular certificate has to cover the three aspects process-based evidences, product-based evidences, and design dependencies. With regard to processes, the system must typically be developed according to the process demanded by the relevant-safety standard. Sometimes, the respective process cannot be performed completely by the developer of the component or the module so that certain activities remain to be performed by the integrator. The same is true for the generation of product-related evidence. Some evidences, such as integration tests, have to be produced by the integrator as well. Both residual product- and process-based evidences must be specified at the certificate interface, so that it is known how to complete the required set of evidences. In addition to that, the evidences already generated by the module developer must typically be provided to the integrator so that the integrator is able to present a compilation of evidences to the certification body. Finally, the interfaces must also contain the safety-critical design-related dependencies (structural as well as behavioral) to other modules, specifying under which conditions the module can be used safely. Our approach focuses only on the design-related dependencies.

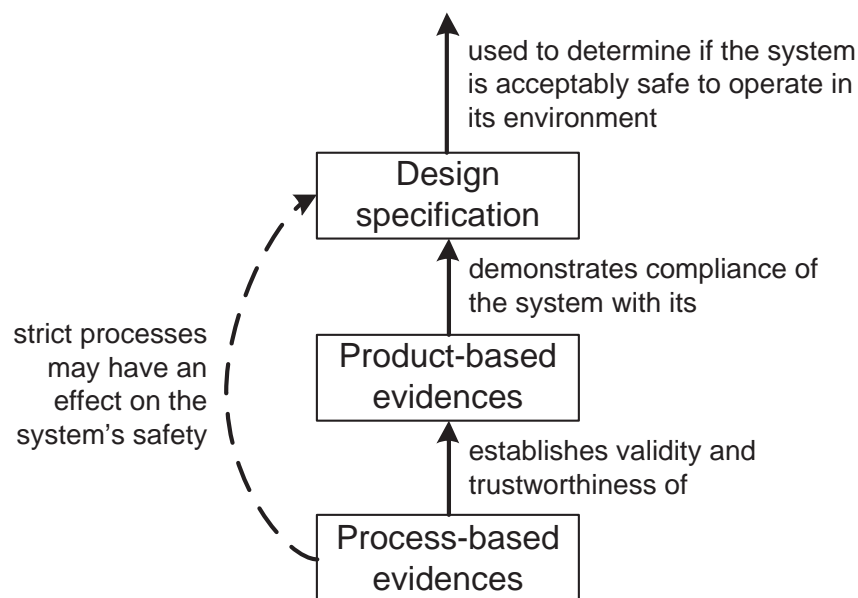


Figure 10: Dependencies between different kinds of evidences in the development-lifecycle of a safety-critical system.

As indicated at the beginning of this section, modular certification is a general term. This means that it does not restrict the product that is

certified modularly. In this thesis, however, we focus on modular certification of applications and platforms and the interface between applications and platforms as described in subsection 2.1.4. To demarcate this interface from other interfaces, we differentiate the *interface orientation*.

A *horizontal interface* describes functional dependencies between two entities on a peer level. In the context of integrated architectures, such interfaces exist between applications. Assume that, for example, the ESC of a car requires knowledge of the car's velocity to operate safely. Consequently, the ESC might demand that the provider of the information indicate whenever the velocity is unreliable or unavailable. Since there are an infinite number of possible dependencies between functions, it is difficult to define a language for describing the dependencies.

In contrast to that, the *vertical interface* describes dependencies between a component implementing a system-level function and a component providing a general, function-independent service. Those function-independent components are typically developed for reuse, and one might refer to them as COTS (Commercial off the shelf). Examples are libraries, communication protocols, or operation systems. The developers of COTS do not know all future systems their product will be part of and they do not know the functions they contribute to. Therefore, it is impossible to perform a hazard and risk analysis for such a component. On the other hand, the function using the general-purpose component knows the kind of service provided by the component and can analyze which failures of the component could cause a hazard. The interface relevant for this thesis, which is the interface between an application and a platform, is a special type of vertical interface.

Coming to the third branch of our classification, the specification of the certificate interface is only the first step towards achieving the goal of reusing certificates efficiently and flexibly. The second work step is the integration of the module certificates into the system certificate. This step includes checking the satisfaction of the mutual dependencies between the module and system, as well as the necessary generation of an argument that the dependencies are satisfied. In our classification, we therefore differentiate between approaches that support interface specification and those that also allow for the efficient integration of the modules. Our method addresses the specification using a formal language (see Contrib. 1) and the integration of modular certificates (see Contrib. 2).

In the following subsections, we will use the classification presented above to classify the related work in the context of modular certification. Since there are many approaches that focus on the process-related

aspects of modular certification, these approaches are summarized in one subsection. All other approaches are described in separate subsections. An overview of the classification of related work is shown in Table 4.

Table 4: An overview of the modular certification aspects addressed by the different approaches in our related work. An "x" indicates that the approach denoted by the column addresses the aspect denoted by the row. An "x" in parentheses indicates that the aspect is addressed only marginally.

Criterion \ Reference		A	B	C	D	E	F	G	H	Our Approach
design dependencies		(x)			x ⁹			x	x	x
	product-based evidences		x	x	(x)	x	(x)			
	process-based evidences	x	x	x	(x)	x				
interface orientation	general vert. interface	x	x	x			x	x		
	app.\plat. interface				x	x			x	x
	horizontal interface						x	x		
working steps	interface specification	x		x	x	x	x	x	x	x
	interface integration	x				(x)		x		x

Table 5: Mapping of identifiers used in Table 4 to references.

ID	name
A	Road Vehicles – Functional safety; Part 10: Guideline; Chapter 10: Safety element out of context [46]
B	AC 20-148 - Reusable Software Components [47]
C	Open IEC 61508 Certification of Products [48]
D	Modular certification support - The DECOS concept of generic safety cases [49]
E	DO-297: Integrated Modular Avionics (IMA) - Development

⁹ Only the specific design of the DECOS platform is addressed.

	Guidance and Certification Considerations [7]
F	The Goal Structuring Notation – A Safety Argument Notation [50] and Architectural Considerations in the Certification of Modular Systems [51] and Safety case architectures to complement a contract-based approach to designing safe systems [52]
G	Boosting Re-use of Embedded Automotive Applications Through Rich Components [53]
H	Safety Analysis of Computer Resource Management Software [54]

Before the presentation of the state of the art, we want to clarify that modular certification is not to be confused with the modular specification of failure logic as done with Component Fault Trees (CFTs) [55], the Failure Propagation and Transformation Notation (FPTN) [56], Hierarchically Performed Hazard Origin and Propagation Studies (HiP-HOPS) [57], or SafeComp Component Model (SaveCCM) [58]. Modular certification evolves around the contract-like specification of demanded requirements and guaranteed capabilities, which may include information about produced, detected, or handled failures, but is not limited to it.

2.3.1 Process-focused Approaches

This section provides a summary of five approaches that mostly address the process-related issues of vertical modular certification. Some of them provide guidelines regarding product-related evidences, too, but only few and coarse-grained specification-related recommendations are given. The first three approaches address general vertical modular certification; the two remaining specifically address open integrated architectures.

ISO 26262 [46] is a safety standard adapted from IEC 61508 [59] to fit the specific needs of the automotive sector. Part ten of ISO 26262 contains non-mandatory development guidelines, and chapter ten of this part introduces guidelines for the development of a Safety Element out of Context (SEooC). The standard defines an SEooC as a safety-critical element¹⁰ for which an item does not exist at the time of its development. This definition complies with our definition of vertical interfaces.

¹⁰ An element is the ISO 26262's term for any kind of part of a larger hierarchical entity, up to the system under development itself. The term was chosen to evade pre-allocated terms like module, component, and such.

The guidelines presented suggest that the developer of an SEooC shall make assumptions about the safety-related properties required from his product and develop it accordingly. When using the SEooC, the integrator shall check and provide evidence that the element as specified fits into the system's safety concept. The standard only provides a very high-level guideline for addressing these process-related aspects. Guidance on how the assumptions should be specified by the developer or checked by the integrator is not given.

In the aviation domain, the Federal Aviation Administration (FAA) advisory circular (AC¹¹) 20-148 [47] describes a process that can be used to acquire *acceptance* for a reusable software component to be used in airborne systems. An *accepted* component can be used more easily as part of a safety-critical system, since certification-relevant artifacts can be reused in the context of the system's certification. AC 20-148 specifically deals with software components that are a part of an airborne system's software application but might not be a software application by itself. As examples, the AC names libraries, operation systems, or communication protocols. This puts the circular in the class of general approaches for vertical modular certification.

The core idea of AC 20-148 is that the developer of a reusable software component may fulfill only a subset of the objectives, or partially fulfill single objectives of the RTCA/DO-178B¹² [60]. The integrator may reuse the credit of the fulfilled and partially fulfilled objectives, and perform the remaining activities (e.g., integration tests) to comply with the residual objectives. The circular focuses on guidelines for complying with RTCA/DO-178B in a distributed and reuse-centered context, which is why we classify the approach as process-related. However, there are some specification-specific aspects that have to be defined by the developer and checked by the integrator. These include features of the component, such as error detection or partitioning, as well as constraints for the use of the component, such as certain hardware failures that the component cannot detect or control.

Open Certification [48] is an application-domain-independent method for vertical modular certification in the context of IEC 61508 [59]. The method is designed to produce two deliverables. First, the safety case provided to the certification body and second, an open document provided to the integrator of the product.

¹¹ An AC never contains mandatory instruction, but advice. In this case, the AC provides one, but not the only, possible means for developing reusable software components.

¹² RTCA/DO-178B is the most common standard for the certification of safety-critical software in airborne systems.

The safety case contains a list of all requirements demanded by the IEC 61508 and either evidence for each requirement's fulfillment, a rationale why the requirement is not applicable to the specific product under development, or why the requirement has to be fulfilled by the integrator. The open document is called safety manual and contains the information that is relevant for the integrator. Besides the remaining process activities for the integrator, the safety manual also contains probabilistic data like MTTF, maintenance requirements, and restrictions for the safe application of the product. The method is process-heavy but provides a distinction between the realization of the safety-critical module (safety case) and its interface specification (safety manual). Using this separation, the developer is able to protect intellectual property from the customer while allowing the customer to integrate the product into the system safety case.

The EU project DECOS (Dependable Embedded Component and Systems) developed an eponymous open integrated architecture for safety-critical embedded systems [61]. The method described in [49] describes an approach for vertical modular certification in the specific context of the DECOS architecture.

The method proposes modularization of the system's safety case into two generic and reusable safety cases for the platform (one for the core services and one for higher-level services) and several application-specific safety cases. Besides this methodological aspect, the authors describe the specifics of the safety cases for the DECOS platform. The approach does not provide any guidelines for checking the sufficiency of the platform's capabilities when faced with different kinds of applications and does, therefore, not provide any guidance on integrating the application and platform safety arguments. The general idea of splitting application and platform safety cases aligns well with the approach pursued by our method. Furthermore, the DECOS safety case can be used for evaluating the specification language of this thesis, comparable to the safety-related feature specifications of AUTOSAR and ARINC 653.

The safety standard RTCA/DO-297 [7] contains guidelines for the certification of systems in the context of Integrated Modular Avionics (IMA). The document describes processes for gathering *incremental assurance* for the certification of an IMA system. The overall process is called incremental acceptance and allows obtaining certification credit for modules, platforms, and applications. Regarding applications, the standard allows their certification on a specific platform, independent from other applications if a robust partitioning is guaranteed. An independent certification of application and platform, however, is not supported.

There are four different tasks in the specified incremental acceptance process. Task 1 allows the modular acceptance of certain parts of a

platform, called modules, and of the complete platform. Task 2 describes the process for accepting applications in an IMA system. An application may only be modularly accepted together with its future IMA platform, but without considering the other applications hosted on the platforms. Subsequently, these applications are incrementally integrated into an IMA system in task 3. Task 4 deals with the integration of the IMA system into the aircraft.

Since initially, applications may only be accepted together with the host platform, a modular specification of the vertical demands of the application is not needed. The standard requires the application developer to check whether the health monitoring and fault management services of the platform are sufficient; a more in-depth guideline is not provided. Task 6 specifies the demands for reusing an application in the context of another platform, but mostly refers to the previously mentioned AC 20-148 and gives little further guidance.

2.3.2 Modular GSN

The Goal Structuring Notation (GSN) [50] is a graphical notation for modeling safety cases. The same paper provides a widely accepted definition for the term *safety case*:

“A safety case communicates a clear, comprehensive and defensible argument that a system is acceptably safe to operate in a particular context”

The argument contained within a safety case is meant to be used to convince third-party assessors. With GSN, the argument is composed hierarchically in a tree-like structure. A typical GSN architecture looks as follows: The top part of the tree contains the hazards that have to be controlled or the safety goals that have to be reached, respectively. Via a logical chain of argumentation and via several layers of sub-goals, the top-level goals are connected to product- and process-related evidences in order to substantiate their fulfillment.

To cater for the modularity of open integrated architectures (especially for IMA), [51] proposes an approach for modularly constructing safety cases with the GSN. To this end, the approach allows specifying a modular interface for a safety case, which may comprise goals, evidences, or context definitions provided (outgoing) or needed (incoming) by other modular safety cases. According to [51], the idea of such an interface is comparable to the rely-guarantee approach, where the provided goals (analogously evidences or context definitions) of a safety case are guaranteed to be fulfilled only if the required goals of the safety case are fulfilled, too. Besides the graphical notation, [52] describes guidelines for the design of modular safety cases in general.

The guidelines for the design of modular safety cases and the idea of specifying design dependencies between modular certificates in a rely-guarantee like fashion provide a basis for our methods. In our approach, we make use of this more generic idea, and tailor it to the specific needs of the application/platform interface.

2.3.3 Rich Component Model

The Rich Component Model (RCM) [53] is a methodology for designing embedded systems that focuses on modularity. RCM allows the definition of module interfaces using a contract-like assume/guarantee semantics. The interface specification of a module comprises several views, covering functional as well as non-functional aspects including a safety view. Furthermore, RCM supports the definition of horizontal as well as of vertical interfaces.

The RCM methodology does not explicitly provide a language for the specification of module interfaces, but most of the examples found use an automaton-based specification approach [62], [53], [63]. It seems that the focus of RCM does not lie on the specification language, but on more general formalization of interface contracts. The methodology contains formal specifications for logical operators to specify, for example, the union or the intersection of multiple guarantees or demands of a single module.

Considering module integration, the approach contains formal specifications of operators for composing interfaces. Checking the fulfillment of properties over a set of composed interfaces is non-trivial and specific for the specification language chosen. [62], for example, propose an approach using formal verification for a hybrid automata specification language.

RCM is not tailored to a particular specification purpose. This versatility usually requires a very powerful specification language, like hybrid automata, to be able to model the variety of embedded systems. The specification language proposed in this thesis, however, is a language tailored specifically to the particularities of the vertical interface between applications and platforms. Complicated dependencies between applications and platforms that would have to be modeled using hybrid automata are available directly in our method.

2.3.4 Safety Analysis of CRMS

The method presented in [54] allows performing a modular safety analysis of Computer Resource Management Software (CRMS). The author defines CRMS as *“any software whose main function is to provide dedicated software with access to generic computer hardware*

resources". We adopted this definition in our work, but referred to the term CRMS as platform software. The developed safety analysis method uses the guideword-based techniques SHARD [64] and LISA [65], as well as patterns describing typical platform software services like communication or scheduling.

Furthermore, [54] describes how to specify the results of the modular safety analysis. The technique for capturing the analysis results is again contract-based. Since the author found that there are dependencies between platform software and applications on several levels of abstraction, the specification technique allows the specification of contracts on three levels: the architectural level, the behavioral level, and the performance level. On each level, the technique includes the specification of guarantees provided by the platform software and demands that have to be met to validate the respective guarantees.

Finally, the method contains a two-step process describing how to use the results of such an analysis for the development of a safety-critical application that uses the analyzed platform. The first step requires the application developer to show that the application fulfills the demands specified by the platform developer. The second step of the process describes how to integrate the platform guarantees into a system-level safety case in order to show how they help to control the function-specific hazards.

The most obvious distinguishing feature between the method proposed in [54] and our method is that our method assumes that platforms as well as applications are developed modularly and are integrated by a third party, whereas [54] assumes that only the platform is reused. If the application is to be reused, this entails the need for a modular specification of the demands of the application in addition to modular specification of the guarantees and demands of the platform. As a second point, this thesis deals with the platform as a whole, consisting of platform hardware and software, whereas platform hardware is explicitly excluded in [54].

Apart from these two distinguishing points, we incorporated some ideas described in [54], like the analysis technique and the contract-based specification of dependencies. Along these lines, we developed a more formal way of specifying the dependencies between application and platform and an automated integration process for capitalizing on this formalization. The method for *"performing a modular safety analysis of Computer Resource Management Software"* introduced in this subsection is completely based on natural language, whereas the Rich Component Model introduced in the previous subsection is completely formalized. Our method, on the other hand, takes an approach in the middle between natural language and formalization to reach a trade-off between automation and ease of use. This trade-off will be discussed

briefly in chapter 3 and in more detail in chapter 5, when our technique is introduced in its entirety.

2.3.5 Conditional Safety Certificates

Open integrated architectures allow for a more flexible and dynamic system architecture, which calls for modular specification of safety cases and efficient methods for integrating these safety cases at the system level. In open integrated systems, however, we have to deal with architectures that change dynamically during design time, especially during the deployment phase. Current trends such as ubiquitous computing or cyber-physical systems will, however, produce “open” embedded systems, like harvesting fleets combined from multiple architectural vehicles or car2car applications. Those systems are characterized by having an architecture that might even change during runtime.

Nevertheless, if an open embedded system is safety critical, there is a need to provide sound safety assurance. The idea behind the Conditional Safety Certificates (ConSerts) approach [66] and the Runtime Certification approach [67] is to shift parts of the safety assurance to runtime when all elements of the architecture as well as their capabilities and requirements are known. ConSerts allow establishing predefined modular safety certificates for each entity of the future system. Those certificates are conditional in the sense that they define requirements on the behavior and the capabilities of other entities as a condition before safe behavior is guaranteed. When each entity provides such a modular runtime certificate, an algorithm is able to check at runtime whether the current system combination fulfills the predefined constraints and is therefore able to run safely.

ConSerts and our approach have the modular specification of certificates as well as the automated integration check in common. ConSerts are, however, different in two aspects. Besides being developed for runtime application, ConSerts operate on the horizontal level between different applications, whereas our approach modularizes the certificates along the vertical axis between applications and platforms.

3 Solution Overview

In the previous chapter, we gave an overview of the current state of the art in developing safety-critical open integrated systems, describing the modular development of applications and platforms and their subsequent integration. Following the presentation of existing approaches as well as the remaining challenges in this field, we will now present our solution for solving the challenges claimed by our contributions with this chapter. For reasons of clarity and comprehensibility, we have divided the detailed description of our solution into three parts and will describe these separately in chapter 4, chapter 5, and chapter 6. In this chapter, however, we will present the different parts of our method jointly and will elaborate on their relations and interactions in order to reach our overall goal of “*efficiently deploying safety-critical applications onto open integrated architectures*”.

Since our approach addresses particular challenges introduced by the development of open integrated systems, we introduce our solutions in the context of the respective development process. An initial version of that process was already sketched in subsection 2.1.1 (Figure 3). However, the process model presented in chapter 2 gives a more general overview of the development process of open integrated systems. Since our work focuses on safety, we need to have a closer look at the safety-specific process steps before we can summarize the core idea behind our work. Therefore, we will discuss a more safety-focused process in this section, as depicted by Figure 11.

The new process separates the development lifecycle of applications and platforms into a product engineering activity and a safety engineering activity. The application safety engineering activity produces one major artifact that is relevant for our consideration: the *vertical safety interface of the application*. The need for this particular artifact is due to the stringent modularization of applications and platforms and contains the demands on the safety-related behavior of the platform that are necessary to argue the safety of the application in a modular way, i.e., isolated from the platform. With the demands specified in this interface, the application developer is capable of arguing the soundness of the application safety case under the assumption that the specified demands are fulfilled. Comparable to the application development, the platform developer produces a *vertical safety interface for the platform*. Complementary to the application safety interface, which contains the assumptions on the safety-related capabilities of the platform, this

document specifies the actual safety-related capabilities that the platform provides towards the application.

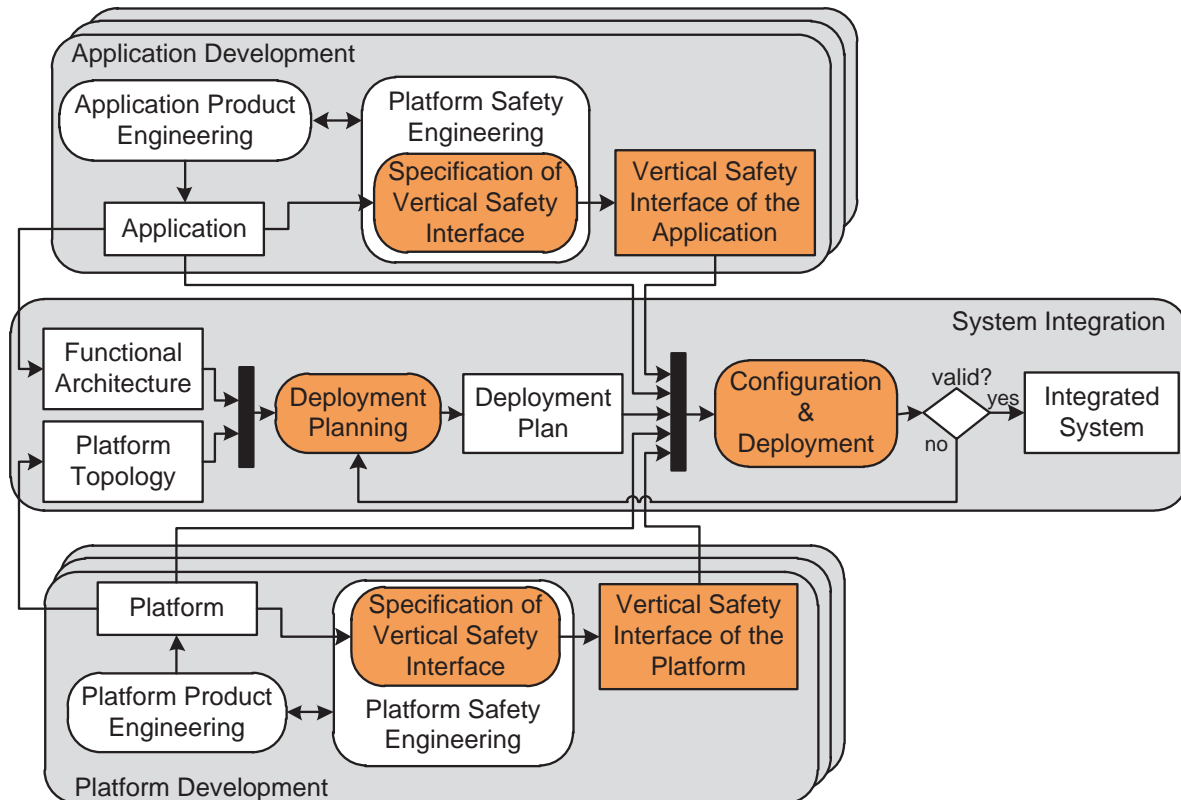


Figure 11: A more detailed version of the OIA development process already illustrated in Figure 3. Orange-colored elements mark the activities and products affected by our contributions.

The idea of separating safety case and safety interface is comparable to the separation of realization and specification well known from component-based development. There, the specification of a component describes its required and provided services while hiding the realization of the services. Analogously, the safety interface describes the safety-related guarantees and demands of the product while hiding the detailed design information, the arguments, as well as the evidences contained in the safety case. The specification of safety interfaces for modular development of safety-related systems is not new to our work. Instead, it is a widely accepted practice in academia and, to a certain extent, also in industry (see section 2.3 for more information on related work w.r.t. modular certification).

A novelty, however, is our specification language for Vertical Safety Interfaces, or VerSal language for short. The *VerSal language* allows application and platform developers to specify their safety case interfaces in a model-based and formal manner. The formality of the specification is a prerequisite for the tool-supported and automated integration of safety interfaces on the system level, which is the final goal of our method. The VerSal language implements the first contribution initially

introduced in chapter 1, which is repeated here for reasons of convenience:

Contrib. 1 **Interface Specification:** *Defining a formal language for the modular specification of safety-related demands and guarantees between an application and a platform in an open integrated architecture.*

When an application is finally mapped onto its execution platform, the system integrator uses the vertical safety interfaces of the application and the platform to check whether the given application is capable of executing safely on its specified host platform. This integration process is where the second part of our solution comes into play. The *VerSal mediator* combines the vertical safety case interfaces specified with the VerSal language and checks whether the demands of the applications can be fulfilled by the capabilities of the corresponding platform. This mediation process is mostly automated and implements the second contribution provided by our approach:

Contrib.2 **Interface Mediation:** *Developing an automated process for checking the safety compatibility of an application and a platform in an open integrated architecture.*

The *VerSal language* for interface specification will be introduced in detail in chapter 4, whereas the *VerSal mediator* for interface mediation will be presented in chapter 5. Yet, in the next subsection (3.1), we will describe the relationships between both components and provide a more detailed description of how the *VerSal method* interfaces with the development lifecycle.

As mentioned before, the VerSal mediator assumes that the modular developed applications and platforms are already mapped to each other and, as shown later, this mapping information is a key input to the automated mediation provided by the mediator. The system integration step that determines the mapping of applications to platforms is called deployment planning (see section 2.2 for more information). To assist in the identification of suitable mappings / deployment plans, we developed an objective function that can be used for *deployment evaluation* and consequently for *deployment optimization*. This objective function corresponds to our third contribution.

Contrib. 3 **Deployment Evaluation:** *Developing a metric for evaluating the deployment of a functional architecture onto a platform topology from a safety perspective.*

In the section after the next (3.2), we will give a short overview of this contribution; a detailed description of the deployment evaluation will be given in chapter 6.

3.1 Interface Specification and Mediation

In this section, we give a summary of the process for the specification of vertical safety interfaces using the **VerSal language** and the consolidation of the resulting interfaces using the **VerSal mediator**. The overall approach is called the VerSal method and is sketched in Figure 12.

In the first step of the VerSal method, the application developer specifies the vertical application interface using the VerSal language and the platform developer specifies the vertical platform interface with the VerSal language. The VerSal language contains different classes of language elements that allow for the specification of different kinds of safety-related dependencies between an application and a platform. It allows the application developer, for example, to demand the detection or avoidance of typical platform failures, but it may equally demand the provision of certain monitoring or failure reaction mechanisms by the platform. When an application or platform developer specifies a safety-related dependency, the developer selects one of these predefined classes and instantiates and configures the class to his or her needs.

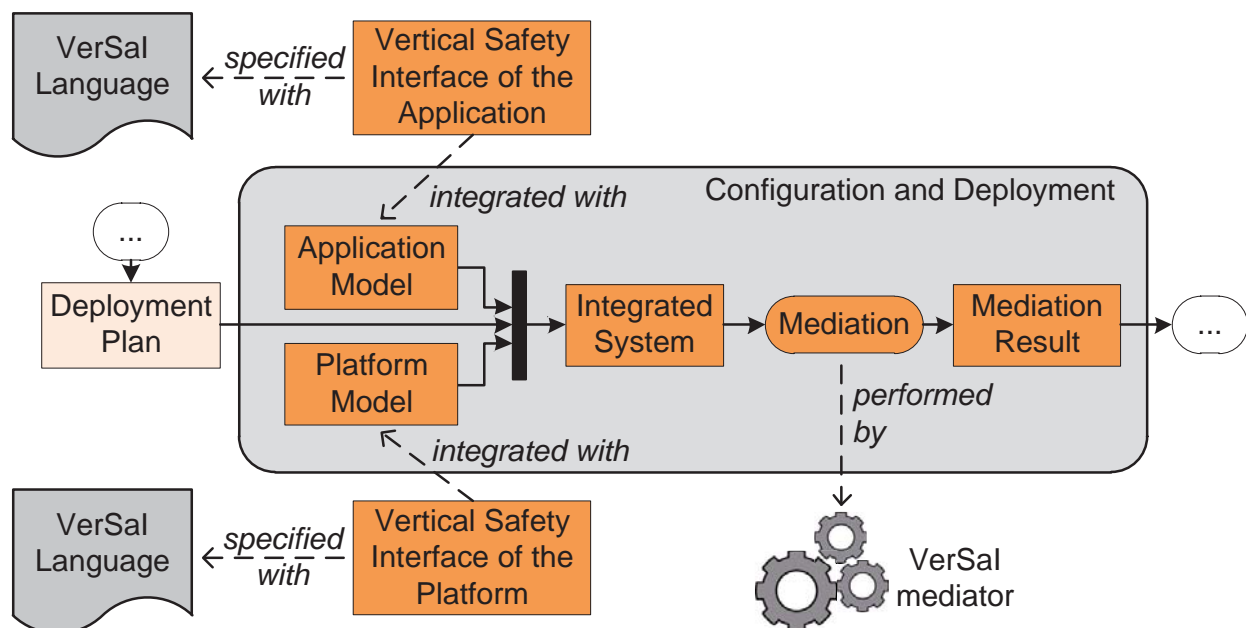


Figure 12: Overall process of interface specification and interface mediation with the VerSal method.

Example 1 shows an example application demand regarding the detection of a value failure of an analog output signal called **a_set_fin**. Example 2, on the other hand, presents the corresponding guarantee provided by an analog output channel called **voltage_out**. The examples showcase certain parameters like failure mode, maximum deviation, failure detection time, or integrity level, which can be variably

configured by the user. To mark these parameters, we underline them in the following examples:

Example 1: A value failure of the output signal a_set_fin larger than 0.05V must be detected within 0.05ms (ASIL C).

Example 2: A value failure of an output signal issued via voltage_out larger than 0.02V is detected within 0.03ms (ASIL C).

The examples allow us to observe another core feature of the VerSal language, namely the integration of the language into the model-based design artifacts of applications and platforms. Referring to the examples presented above, the application demand (example 1) is linked to the model-based representation of the corresponding signal called **a_set_fin**, and the platform guarantee regarding corruption detection is linked to the model-based representation of the corresponding output channel called **voltage_out**. This integration of safety model and design model facilitates consistency between the safety model and the design model, enables certain plausibility checks, and, most importantly, is required by the VerSal mediator to perform the automated interface mediation.

Once the vertical interfaces of application and platform have been specified, the aforementioned VerSal mediator checks whether the application demands can be fulfilled using the available platform guarantees. During this step, the integration aspect of the VerSal language is used to match demands with their relevant guarantees. To perform this matching, the mediator uses an additional piece of information – the deployment. The deployment is an integral part of a regular development process and specifies the mapping of application elements (e.g., of output signals) to platform elements (e.g., output channels). Since the demands and guarantees of the VerSal language are related to their corresponding language element, the deployment information can be used to match a demand with the relevant guarantees in a transitive fashion. The mediator navigates from a demand to the related application element, via the deployment information to the corresponding platform element, and finally from there to the relevant guarantees.

If the guarantees that are relevant for the fulfillment of a demand are found, the next step in the mediation process is to analyze whether the demands can be met by these particular guarantees. The detailed process describing the mediation of the different demand classes provided by the VerSal language is described in chapter 5. The VerSal language itself is described in chapter 4.

3.2 Deployment Evaluation

In the last section, we gave an overview of the process for specifying and checking the fulfillment of safety-related dependencies between applications and platforms using the VerSal method. One of the required inputs for the method is a deployment plan specifying the mapping of the applications onto the platforms of the system. In this section, we will describe how our third contribution, the objective function for deployment evaluation, helps to find a suitable deployment plan.

The first challenge in identifying potential deployment candidates lies in the size of the solution space. If we only regard the assignment of applications to platforms, this results in p^n possibilities, assuming that there are n applications and p platforms. To handle the high number of possibilities, the evaluation of the solutions is usually performed automatically. The second challenge lies in the variety of design criteria that influence the quality of a deployment plan. As we have shown in chapter 2 (subsection 2.2), there are also conflicting criteria that are hard to weight and compare with each other. In the context of this automated multi-criteria deployment evaluation, we developed two novel metrics for deployment evaluation.

The *cohesion metric* focuses on the aspect of unprotected shared computational resources in a mixed-critical system, as the metric evaluates the costs of interferences between ASWCs. The *coupling metric*, on the other hand, evaluates the costs caused by safety mechanisms to protect against communication failures, which are incurred when separating tightly coupled components over the platforms of a distributed system. It is important to note that in a real-world application, our metrics have to be used together with other objective functions, since there are multiple other quality criteria that have to be evaluated as well.

Figure 13 illustrates the process of deployment optimization that involves our metrics. The solution space of the deployment is defined by the functional architecture specifying the elements that have to be deployed and the platform topology specifying the target environment of the deployment. The cohesion metric, the coupling metric, and some additional constraints that can be specified using our approach are used together with other objective functions and constraints to identify the solution candidates. The cohesion metric, the coupling metric, and the deployment constraints that can be modeled with our approach will be introduced in chapter 6.

The deployment plan that is generated by this optimization specifies a relatively high-level mapping of applications to platforms. It is important to note that there is a manual step involved that refines this high-level

mapping before it reaches the level of detail required for the VerSal approach¹³.

As a final comment to this overview, we want to remark that there are many approaches for optimization, such as Linear Programming [68] or Genetic Algorithms [69]. Our contribution, however, does not focus on the choice of the optimization algorithm but rather on the objective functions for evaluating deployments. However, we have evaluated our metrics with an example in the context of an optimization framework using genetic algorithms, which will also be discussed in chapter 6.

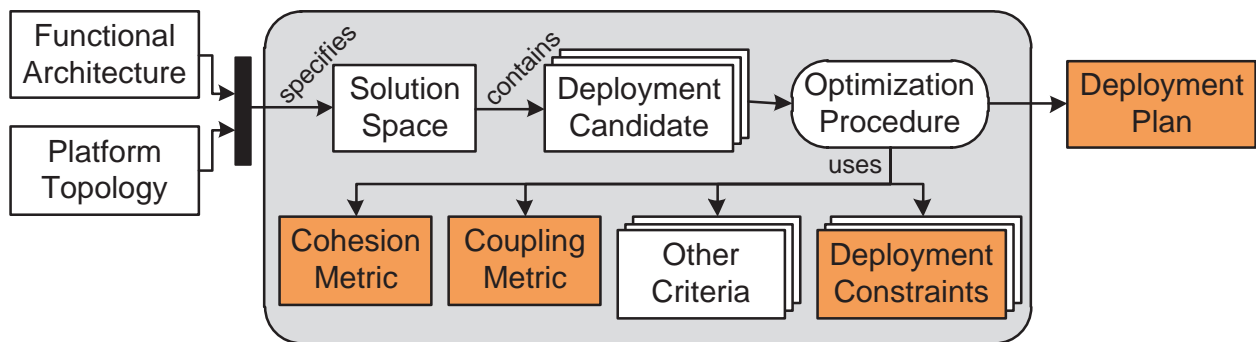


Figure 13: The process of Solution Candidate Identification : Orange-colored elements mark the activities and products affected by our contributions.

¹³ The difference between the high-level and low-level deployment model is best observed by comparing the deployment as specified in Figure 73 with the deployment as specified in Table 16

4 Interface Specification

This chapter is the first of three chapters describing our methods for “Efficiently Deploying Safety-Critical Applications onto Open Integrated Architectures”. In this particular chapter, we describe the realization of our formal language for specifying the safety-related dependencies between applications and platforms, which corresponds to the first contribution specified in chapter 1.

Contrib. 1 **Interface Specification:** *Defining a formal language for the modular specification of safety-related demands and guarantees between an application and a platform in an open integrated architecture.*

Prior to describing the structure of the language and of this chapter, we will summarize the role of the VerSal language in the overall context of our work. The *VerSal language* is part of the *VerSal method* and is used by application developers and platform developers to specify the *vertical safety interface* of applications and platforms, respectively. The VerSal language allows specifying the vertical safety case interface specified in a model-based representation. The VerSal language has been developed with the idea of containing sufficient information so as to allow the *VerSal mediator* to decide (or support the decision-finding) about whether a particular application can be executed safely on a particular execution platform. Figure 14 depicts this role of the VerSal language. It shows the transition from the language specification to the mediation phase, which begins with the configuration of the existing applications and platforms. However, the mediation is beyond the scope of this chapter and will be presented in chapter 5, “Interface Mediation”. In this chapter, the focus is on the description of the VerSal language¹⁴.

The structure of this chapter mirrors the top-level architecture of the language. The VerSal language is divided into three main packages (Figure 15): the *common language*, the *application language*, and the *platform language*. The common language defines types, properties, and relations that are used across the application- and platform-specific language parts. The application-specific part is used by the application developer to specify the demands regarding the behavior of the platform, whereas the platform-specific part is used by the platform developer to specify the guarantees regarding the behavior of the platform.

¹⁴ Please note that a complete overview of the VerSal technique is given in the previous chapter in section 3.1 (for a quick overview, see Figure 12).

This chapter is structured as follows. We begin with the introduction of a running example that will be used across the upcoming sections to exemplify the language concepts. The description of the language starts in section 4.2 with an explanation of the key high-level design decisions that have a cross-cutting influence on most components of the VerSal language. In Section 4.3, we introduce the first part of the common language containing types, parameters, and relations. Section 4.4 describes the second part of the common language, which contains the very core of the VerSal language, the common set of failure modes and failure reaction measures used to specify demands and guarantees. Following this, the application-specific part of the language is introduced in section 4.5, while the platform-specific part of the VerSal language is presented in section 4.6.

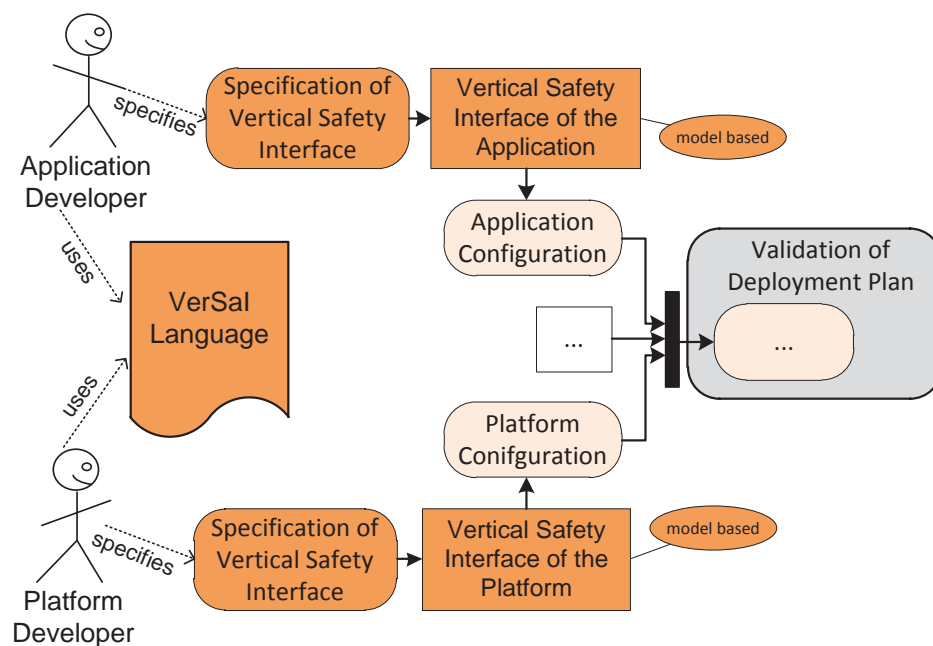


Figure 14: The role of the VerSal language in the VerSal method

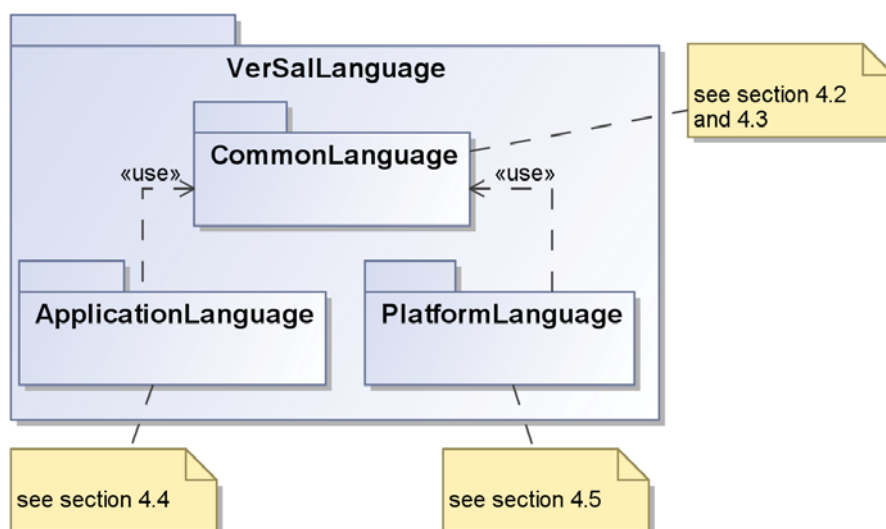


Figure 15: This figure shows the top-level architecture of the VerSal language

4.1 Running Example

In this section, we introduce a running example that will be used across the upcoming sections and chapters to exemplify the concepts of the VerSal language and the VerSal mediator. The running example consists of two parts: an example application shown in Figure 16 and an example platform shown in Figure 17. The mapping/deployment of the application to the platform is shown in Table 6. For more information regarding the underlying meta-model, please refer to Appendix A¹⁵.

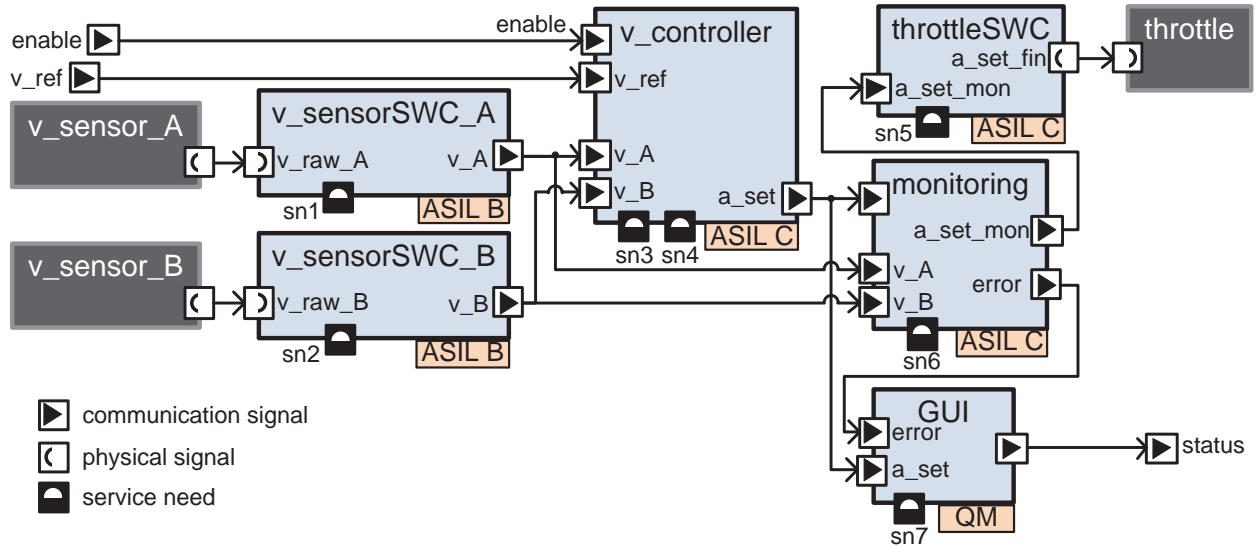


Figure 16: An example cruise control application : Software components are shown as blue rectangles, actuators and sensors as black rectangles. The integrity level of software components is shown in a red tag in the lower right corner of the rectangle.

The example application is an automotive cruise control application. The goal of the application is to control the vehicle's velocity to match a user-defined reference velocity (v_{ref}). The main controller component in our example is called **v_controller**. The controller reads the current velocity of the car, compares it to the reference velocity, and calculates a new set value for the throttle to match the future velocity with the reference value. The current velocity of the car is provided by two redundant sensors (**v_sensor_A**, **v_sensor_B**) and two sensor software components (**v_sensorSWC_A**, **v_sensorSWC_B**). On the actuator side, the acceleration is controlled by a single software component (**throttleSWC**) and a single actuator (**throttle**). The redundant sensor components are used by the application to tolerate single sensor failures. Such a failure is detected by the monitoring software component, which, upon detection, manipulates the values sent to the actuator so as to transition the system into a safe state. In

¹⁵ Please note that, for reasons of clarity, the presented graphical representations of the example do not show every detail of the model.

addition to the afore-mentioned components directly involved in the control loop, the application contains another software component called **GUI**, which provides the user with information regarding the current status of the cruise control application.

Please note that the described example application is a mixed-critical application, i.e., it contains software components of different degrees of criticality. The redundantly constructed sensor components are developed according to *ASIL B*¹⁶, whereas the residual components are developed according to the higher ASIL C, with the exception of the GUI component, which is developed according to QM.

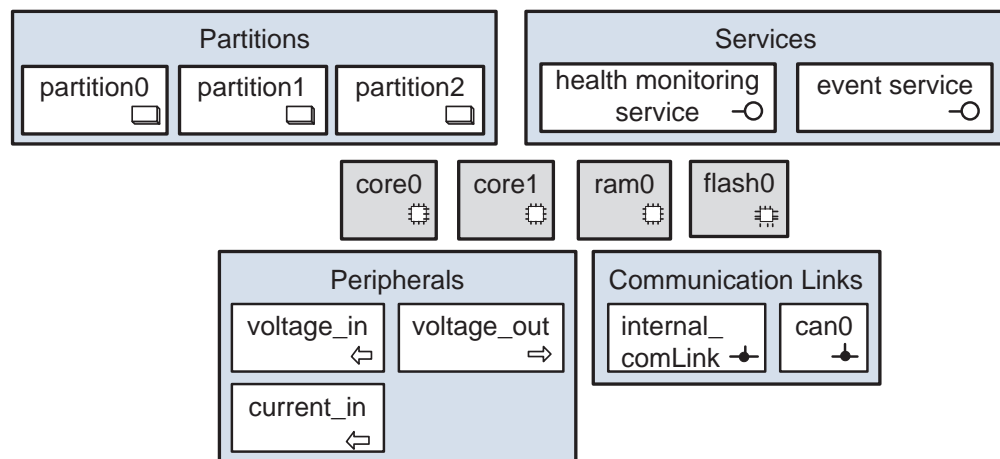


Figure 17: An example platform

The simple example platform hosts an operating system that provides the concept of partitions for separating mixed-critical applications. The platform is currently configured to have three partitions: **partition0**, **partition1**, and **partition2**. In addition to the partitions, the platform offers two software services, a **health monitoring service** and an **event service** for inter-process communication. Regarding peripherals, the example platform offers two input channels, one for reading voltage-based (**voltage_in**) and one for reading current-based signals (**current_in**), and a single output channel (**voltage_out**) for providing voltage-based output signals. Additionally, the platform provides two communication links: an internal communication link to connect software components hosted on the platform (**internal_comLink**) and one communication link for connecting the software components with other platforms (**can0**). Please note that, since we do not further separate platform software and platform hardware, peripherals and communication links as specified in our example consist of software (e.g., the com stack) as well as hardware components (e.g., microcontroller peripherals).

¹⁶ ASIL stands for Automotive Safety Integrity Level. Further information is provided in the glossary.

The deployment of the application to the example platform is shown in the following table:

Table 6:

The deployment of the running example : This is a deployment of the example application onto the resources provided by our example platform.

<i>Resource User</i>	<i>Resource</i>
ASWC : v_sensorSWC_A	Partition : partition 0
ASWC : v_sensorSWC_B	Partition : partition 0
ASWC : v_controller	Partition : partition 1
ASWC : throttleSWC	Partition : partition 1
ASWC : monitoring	Partition : partition 1
ASWC : GUI	Partition : partition 0
SignalInPort : v_raw_A	InputChannel : voltage_in
SignalInPort : v_raw_B	InputChannel : current_in
SignalOutPort : a_set_fin	OutputChannel : voltage_out
ComPort : enable	ComLink : can0
ComPort : v_ref	ComLink : can0
ComPort : v_A	ComLink : internal_comLink
ComPort : v_B	ComLink : internal_comLink
ComPort : a_set	ComLink : internal_comLink
ComPort : error	ComLink : internal_comLink
ComPort : a_set_mon	ComLink : internal_comLink
ComPort : status	ComLink : can0
ServiceNeed : sn1	Service : health monitoring service
ServiceNeed : sn2	Service : health monitoring service
ServiceNeed : sn3	Service : health monitoring service
ServiceNeed : sn4	Service : health monitoring service
ServiceNeed : sn5	Service : health monitoring service
ServiceNeed : sn6	Service : event service.error_event
ServiceNeed : sn7	Service : event service.error_event

4.2 Language Design

In this section, we will discuss the core characteristics of the VerSal language's design. This discussion is structured according to six main characteristics of the VerSal language, which are:

- semi-formal
- finite
- exhaustive
- extensible
- parameterizable
- integrated

The VerSal language must be as *formal* as necessary so that the VerSal mediator can reason about whether the application demands are fulfilled by the given platform guarantees. With our design approach, it is sufficient to formalize the syntax of the language and specify some parts of its semantics informally. The syntax is formalized by the meta-model of the language. Conversely, the semantics of the language elements is specified informally by the descriptions given in this chapter. As we will show in the next chapter, this degree of formalism is sufficient for automatically checking whether the demands and guarantees are compatible.

The basic element of the aforementioned meta-model, and thus of the VerSal language, are safety requirements specified at the interface between application and platform. Each requirement can either be typed as a demand – if it specifies an expectation about the behavior of the platform – or as a guarantee – if it specifies the realization of platform behavior. When we say that the VerSal language is *finite*, we mean that there are a finite number of safety requirement types that can be used to specify the vertical safety interface of a platform or application. Unlike a specification language such as state machines, where a complex semantic whole (the state machine) is composed by putting elementary units into relations (states and transitions), a safety interface specified using the VerSal language consists of a set of independent requirements. On the one hand, this simplicity allows for a mediation algorithm that works with a semi-formal language. On the other hand, however, this means that the expressiveness of the language is limited by the available demand and guarantee types.

To achieve sufficient expressiveness using such a design, the language must be *exhaustive*, which means that every possible safety-related dependency between application and platform must be expressible with the available demands and guarantees. This directly leads to the question of whether it is feasible to enumerate all possible safety requirements

that can occur at the interface between application and platform. We found that, with certain restrictions, this can be done due to one fundamental reason: An execution platform mostly offers standard services. If most platforms did not offer standardized services, one would not have been able to standardize the API of the execution platform in the first place. Based upon those services, we found that a platform and an application share four different classes of safety-related dependencies: (1) Platform Service Failures, (2) Health Monitoring, (3) Service Diversity, and (4) Resource Protection.

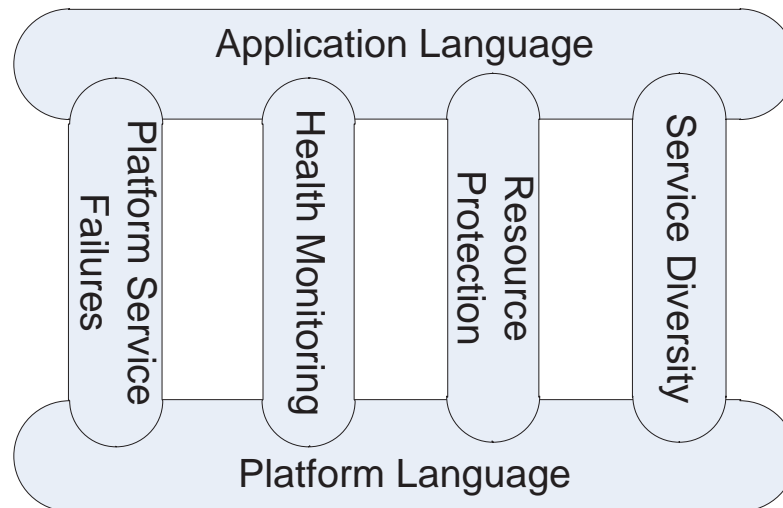


Figure 18: The four classes of safety dependencies in VerSal

The platform needs to provide dependable services to the application in order to enable the application to provide its functions safely. The *platform service failure* class enables the application and platform developer to specify demands and guarantees regarding the detection or avoidance of platform failures that would otherwise affect the safe behavior of the application. In contrast, it is common practice to use the platform as a means for detecting application failures and for executing failure containment reactions. The *health monitoring* class allows specifying the corresponding dependencies. The third class of safety-related dependency is called *service diversity*. It allows specifying demands regarding the diversity of provided services, which is the basis for so-called integrity level decompositions offered by certain standards. The fourth and last class is called *resource protection* and allows specifying demands regarding the protection of services from interference by other applications, which is demanded by most safety standards when mixed-critical applications share common platform resources and services.

Both the application and the platform language are structured according to these four classes and contain corresponding demand and guarantee prototypes. If the developer needs to specify a demand or a guarantee,

the demand or guarantee must be covered by one of the available types for all features of the VerSal language to be available. To deal with demands and guarantees that are not covered, the VerSal language offers two possible solutions. The first is its *extensibility* and the second the possibility to incorporate informal *free-text* demands and guarantees into the interface specification. With regard to demands or guarantees that are not covered by our language, we recommend that a developer specifies the language element using an informal free-text requirement when encountering the issue for the first time. If such a free-text requirement is used, the automated interface mitigation is not available for this particular requirement. Only if a certain kind of free-text requirement is used more frequently, we suggest extending the VerSal method. This task of extending the method is supported by the modular design of the VerSal language and the VerSal mediator.

However, during our evaluation (see 0) we found that the VerSal language is capable of expressing most of the required dependencies between applications and platforms. One reason for this is that the user of this language is able to tailor a demand or guarantee prototype to her or his needs by specifying relations and properties.

The most important relation that can be specified is the safety requirement's architecture references. Architecture references specify the *integration* of the language into the design model of the application and of the platform. If a certain demand or guarantee relates to an architectural element, this element must be referenced by the corresponding safety requirement. As an example, a demand regarding the detection of a signal corruption must reference the corresponding signal (e.g., the signal `a_set_fin` in our example application) or a platform guarantee regarding the detection of corruptions must reference the corresponding communication link (e.g., the `internal_com_link` or `can0` in our example platform). With this information, plausibility tests can be performed and the mediator can automatically put demands and guarantees into relation if the deployment specification is available.

Application demands relate only to elements of the application model and platform guarantees relate only to elements of the platform model. Furthermore, application and platform model are connected with each other via the deployment model (the deployment model can be used to specify deployment plans, see section 3.2). The application model, the platform model, and the deployment model were inspired by existing meta-models like the AUTOSAR or the EAST-ADL meta-model. We present our architecture model in Appendix A. When you read the language specification, you will encounter references to the architecture model. If you consider it important to understand the meaning of certain architectural elements, we recommend looking up the meaning of the

element in Appendix A. The high-level architecture of the VerSal language including the relation to the architecture model is shown in Figure 19.

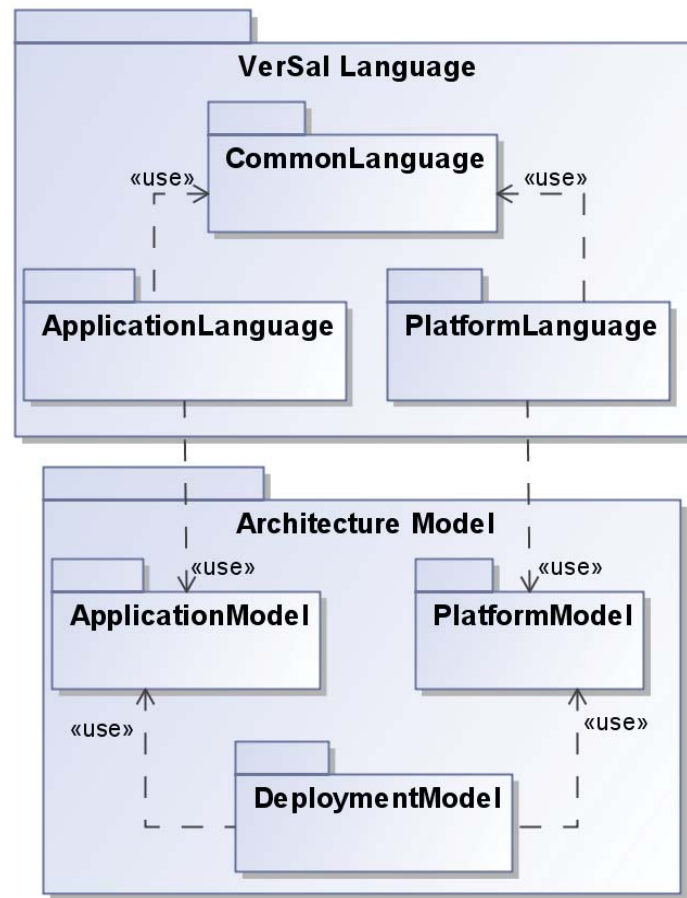


Figure 19: The high-level architecture of the VerSal language and the architecture model

To tailor a safety requirement type during instantiation, each type has certain quantitative and qualitative *parameters*. One mandatory qualitative parameter is the criticality level of a requirement, which classifies the risks caused by not meeting the requirement. Certain requirements also need quantification. If, for example, the application developer demands the detection of a failure mode, the application developer has to specify the fault detection time, which is the time between the occurrence and the indication of the failure.

4.3 Common Language – General Features

This first part of the common language package describes types, relations, and parameters that are used in the application language and in the platform language. Most of these common aspects describe realizations of high-level design concepts that were introduced in section 4.2.

This section will successively introduce the realization of demands and guarantees in subsection 4.3.1 and conditions in section 4.3.2. After that, we will describe the modeling of architecture references in 4.3.3 and that of parameters in 4.3.4.

4.3.1 Demands and Guarantees

The main goal of the VerSal language is to allow *modular* specification of safety-related dependencies between applications and platforms. To allow such modular specification of dependencies, the developers must be capable of specifying assumptions/demands on the behavior of other modules and guarantees regarding a module's own behavior in a contract-based manner. Therefore, the VerSal language employs a demand-guarantee concept for the specification of interface requirements.

A demand is typically used by the application developer when the application development gets to a point where it is impossible to argue about the safety of the application without knowing about the behavior of the platform. At such a point, demands regarding the characteristics of the platform are used to complete the safety argument of the application. With the demands in place, it is possible to assess the soundness of the application safety case without knowing the actual platform, under the assumption that the platform fulfills all demands. When reviewing the safety case, the developer respectively the assessor should come to the conclusion that the "application is safe under the assumption that the application demands are fulfilled by its host execution platform".

A guarantee, on the other hand, is used by the platform developer as a starting point for the safety-related development of the platform. The developer of a general-purpose platform is unaware of the applications that will later run on the platform. Hence, it is impossible to know in advance the detailed demands regarding the behavior of the platform that will be brought forward by the application developer. Therefore, the platform developer must make assumptions about the required safety-related behavior of the platform and develop the platform accordingly. After the platform has been developed and assessed, the assumptions about the required behavior turn into guarantees that can be used during integration to satisfy the demands of the hosted applications.

Consequently, at the root of the VerSal meta-model is the abstract **InterfaceRequirement** class, which is divided into **Demands** and **Guarantees**. Starting from this first classification, the VerSal language is further structured according to the dependency classes introduced in section 4.2. In the leaves of the emerging classification are the instantiable safety requirements that can be used to define a vertical

safety interface. The first three levels of the safety requirement classification are shown in Figure 20. The remainder of the meta-model will be introduced in the corresponding sections. Application-specific refinements will be introduced in 4.5; platform-specific refinements in 4.6.

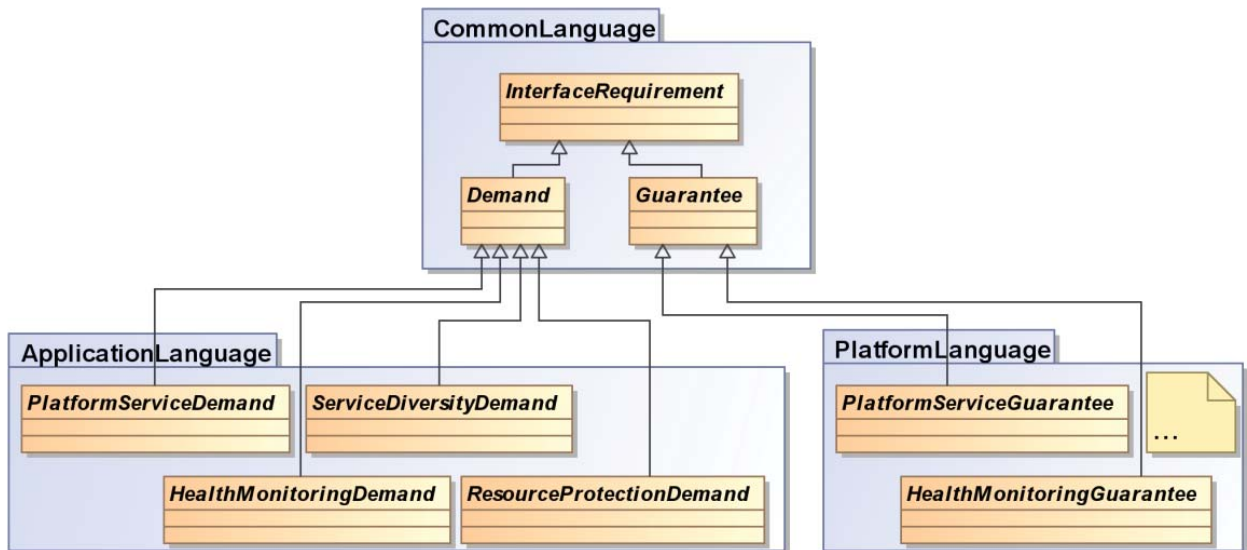


Figure 20: An excerpt of the VerSal language's classification tree

4.3.2 Conditions

Applications and platforms, but especially the latter, are highly configurable and adaptable components. As a consequence, it is often impossible to specify demands or guarantees in an absolute manner, since a demand or guarantee often depends on the configuration of the component. AUTOSAR, for example, provides numerous mechanisms for protecting its operating system, but those mechanisms can be deactivated for performance reasons or do not work as intended for certain configurations. In order to deal with these situations, the VerSal language provides a mechanism for specifying conditions, which is presented in this section.

From a technical point of view, a condition is an expression that the mediation algorithm evaluates as true or false during the mediation process. There are three different kinds of conditions: configuration-dependent conditions (the main use case for conditions), deployment-dependent conditions, and manual conditions. These different kinds of conditions differ only in terms of the information required to evaluate the condition. Configuration-dependent conditions can be evaluated as true or false when the application or platform is configured. Deployment-dependent conditions can be evaluated when the deployment is specified. Manual conditions, however, cannot be evaluated by the mediation algorithm and have to be checked by the

integrator. A condition has one variable called fulfillment that represents the state of the condition. The variable can be set to three states: The variable is set to “unchecked” if the condition has not been evaluated yet. It is set to “fulfilled” if the condition has been evaluated to true, and it is set to “violated” if it is evaluated to false. Figure 21 shows the condition state machine.

The VerSal language currently supports configuration-dependent conditions with an “equals” semantics. The developer of the vertical safety interface specifies a condition by referring to a configuration parameter and by asserting a required value to the configuration parameter. When the integrator has finished the configuration, the mediation algorithm evaluates the condition to true if the corresponding configuration parameter is set to the required values. The VerSal language currently does not support “greater than” or “less than” configuration conditions.

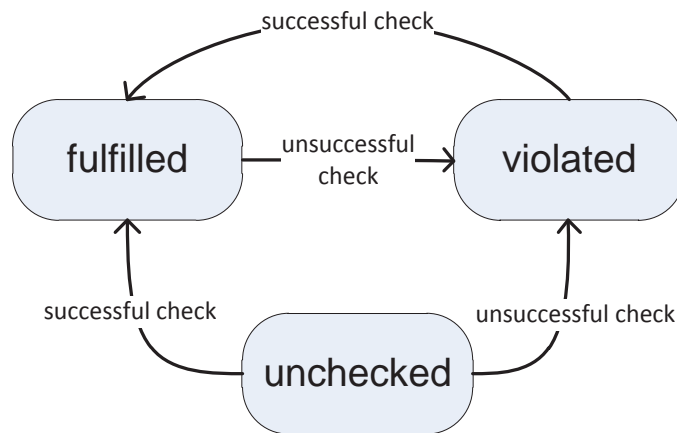


Figure 21: The VerSal condition state machine

The developer specifies manual conditions by describing a condition as plain text and by specifying the evidences that are required to support the condition, if applicable. The integrator can link the evidences that support the manual condition and set the manual condition to true or false. Deployment-dependent conditions are envisaged and rudimentarily implemented in the language and in the mediator, but have not been implemented yet.

Figure 22 shows an excerpt of the meta-model that illustrates the design of conditions.

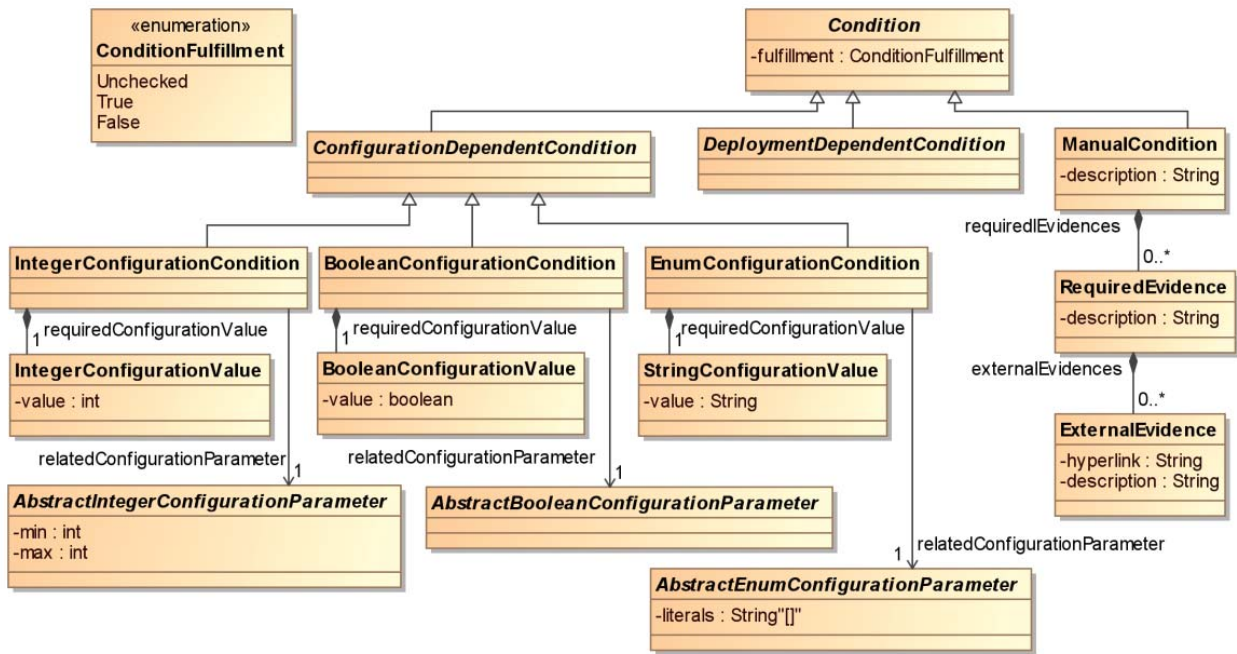


Figure 22: The VerSal condition meta-model

As mentioned above, the primary use case of the condition concept is the specification of conditional demands and guarantees. In order to specify such a conditional language element, the developer is allowed to specify a list of conditions¹⁷. These conditions must be evaluated before the mediation algorithm knows whether the interface requirement is valid or not. For demands as well as for guarantees, all conditions must be evaluated to true before the corresponding requirement is valid. However, the semantics of invalid demands and guarantees differs significantly: An invalid demand makes mediation easier, as an invalid demand does not participate in the mediation, i.e., it is not needed and does not have to be fulfilled. On the other hand, an invalid guarantee hampers the overall mediation as an invalid guarantee is not available and can therefore not be used to fulfill the demands at hand.

Another important use case for conditions is the specification of conditional parameters, which will be introduced in section 4.3.4.

4.3.3 Architecture Relations

Although still a vibrant topic of research, many embedded systems are already being developed using model-based techniques. In fact, some embedded environments like AUTOSAR and ARINC 653 already require a significant portion of the system to be modeled. In order to use the advantages of model-based design, such as automatic generation of

¹⁷ Every type of condition (i.e., configuration, deployment, and manual condition) and every combination of condition types is allowed.

development artifacts, the VerSal language is model-based as well. In order to make use of the information already provided in existing models, the VerSal language provides the architecture relation concept as a plug-in mechanism for connecting the VerSal model with existing architecture models.

Technically speaking, an architecture reference is a link between a vertical safety requirement and the architecture model of the corresponding application or platform. Architecture relations realize the integration of model-based demands and guarantees into the design model of the integrated system. As already described in section 4.1, application demands are always integrated into the application's architecture model and platform guarantees are always integrated into the platform's architecture model.

Architecture relations enable several features of the VerSal language. On the one hand, they foster consistency. If an architectural element changes, the change is directly reflected or indicated at each safety requirement that is related to the architectural element. Furthermore, architecture relations allow for plausibility checks. If, for example, all but one communication port of a software component contain safety-related demands, the remaining port might have been forgotten by the developer. But most importantly, architecture relations enable the mediation algorithm to associate demands with mediation-relevant guarantees. If an application element such as a communication port contains demands, the mediation algorithm is able to identify the related platform element via the deployment information of the communication port, and the related guarantees of the communication link via its architecture relations.

In the VerSal language, there are two kinds of architecture relations: containments and references.

A containment relation allows an architecture element to contain a demand or a guarantee. The components/elements that form the application (application elements) may contain demands and the components/elements that constitute the platform (called platform elements) may contain guarantees. An application element contains a demand if the demand originates from that element: "If the element was not there, there would also be no demand". A communication port, for example, contains demands regarding the detection or avoidance of the signal received via the demand. If we want model a demand regarding the correct reception of the signal `v_ref` in our running example, this demand would be contained in the corresponding port of the software component `v_controller`. A platform element, on the other hand, contains a guarantee if the element is responsible for providing the guarantee. Therefore, an element such as a communication link contains guarantees regarding the control of

communication failures. In our running example, a guarantee regarding the detection of corruptions of the signal `v_ref` is contained in the communication link transporting the signal, in this case the communication link labeled as `can0`.

One of our design goals was to design the VerSal language such that it is minimally invasive regarding the architecture model, i.e., the use of the VerSal language should impose minimal to no changes on the model of the corresponding integrated system. Therefore we chose to automatically generate one abstract class that is contained in every application or platform element. Through inheritance of this abstract class, interface requirements can be flexibly contained in architectural elements without changing the meta-model of the integrated system. An excerpt of the corresponding meta-model is shown in Figure 23.

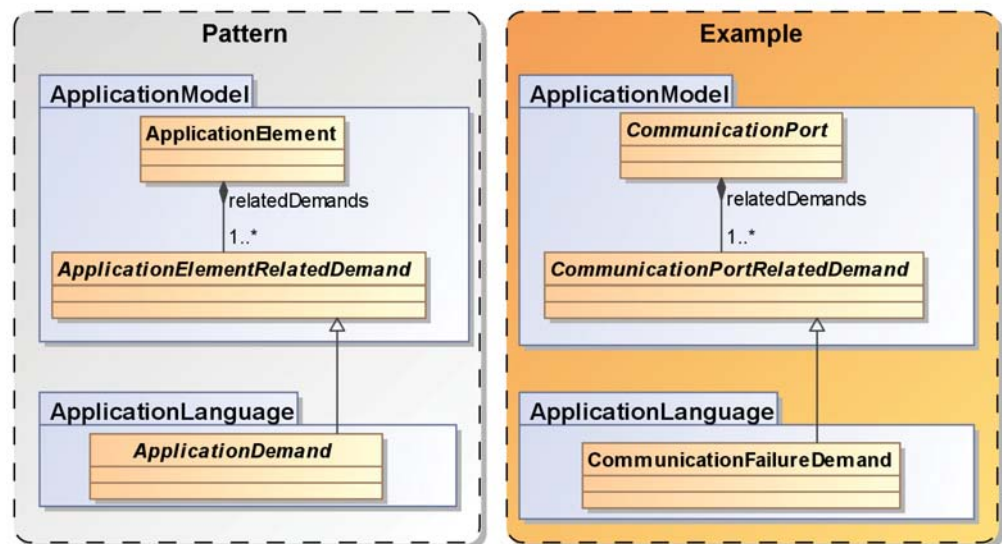


Figure 23: Containment relations in VerSal : The left side of the figure shows the modeling pattern used to realize containment relations in a minimally invasive fashion. The right side shows an example instantiation of the pattern for communication failure demands.

The second kind of architecture relation, i.e., references, are used if a demand or guarantee needs to reference the respective element but the semantic relation between interface requirement and architecture element is not strong enough or is not suitable for a containment relation. Architecture references are generally used if the precise specification of a requirement involves an architecture element but the requirement originates from a different element (which is realized by a containment relation). If there is, for example, a requirement that demands an output port to send a fail-safe signal when a failure has occurred, the affected output port is referenced via an architecture reference. As an example use case let us assume that the software component `throttleSWC` has to deal with scheduling failures. If the component is not scheduled in time, the throttle actuator is not controlled appropriately, which results in a critical situation. Therefore,

the software component demands that the platform automatically sets the signal `a_set_fin` to its fail-safe signal (no throttle demand) in case the component misses its deadline. When specifying such a demand, the demand is contained in the `throttleSWC` component but references the `a_set_fin` port.

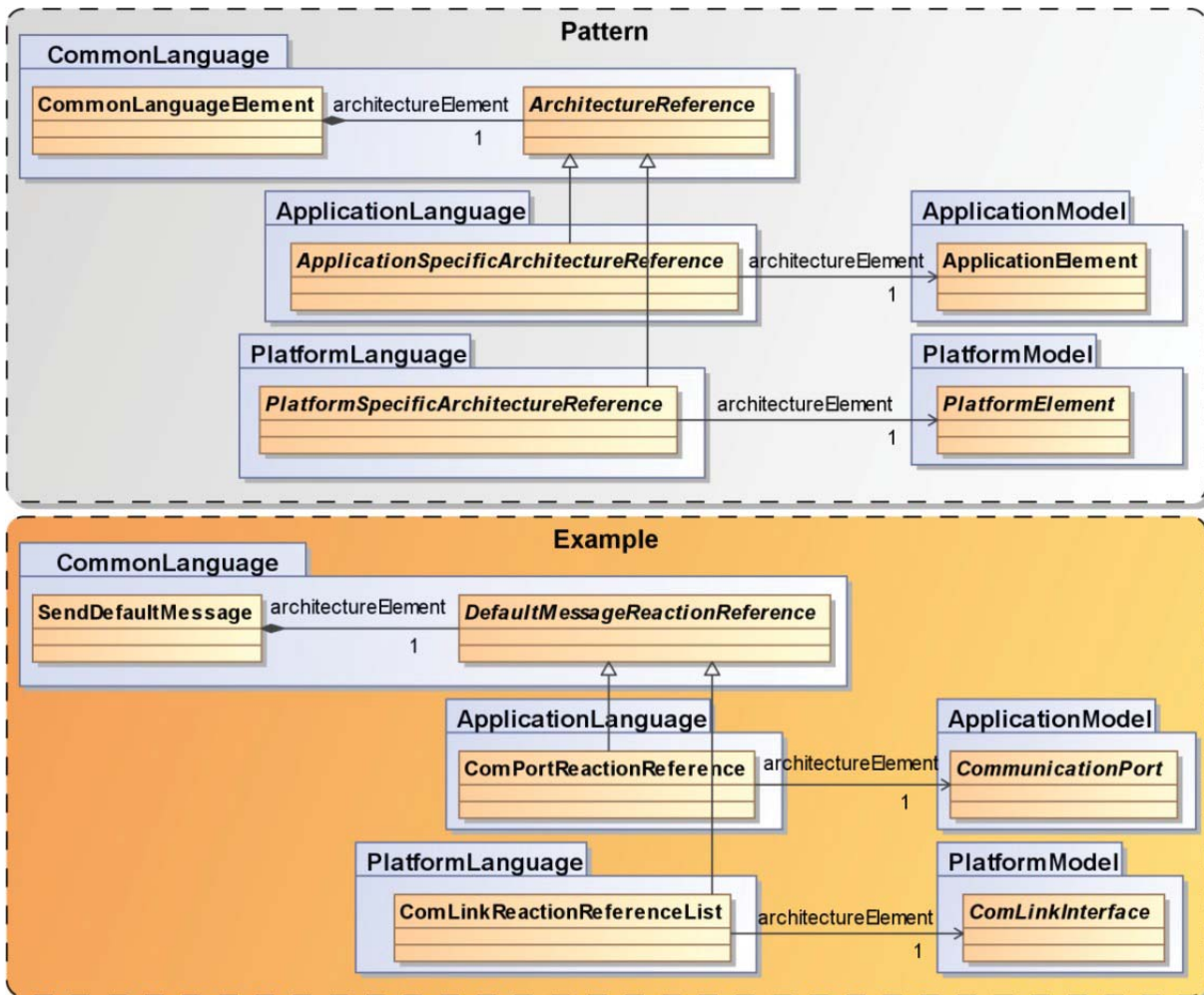


Figure 24: Reference relations in VerSal : The left side of the figure shows the modeling pattern used to realize reference relations. The right side shows an example instantiation of the pattern for referencing ports or communication links that need to output a default message.

However, most of the architecture references do not directly originate from a demand or guarantee, but rather from an element of the common language. The challenge with those references is that, depending on whether the common language element is used in the application language or in the platform language, the target of the reference changes. A send default message reaction, for example, references a communication port in the application language and references a communication link in the platform language. Therefore, the meta-model of the language uses a modeling pattern that is similar to the pattern used for containment relations. The common language element contains an abstract class, which is inherited by different classes

in the application and platform language to realize different relation targets from application and platform. The corresponding pattern is depicted in Figure 24. Please note that the modeling pattern is again minimally invasive regarding the meta-model of the integrated system, as the meta-model is left unchanged.

4.3.4 Parameters

The demand and guarantee prototypes provided by the VerSal language have to be parameterized to adjust the prototypes for the needs of the individual systems. VerSal language parameters are always set during the instantiation of a safety requirement. Example parameters are: the failure detection time of a detection demand, the integrity level of a demand or a guarantee, or the tolerable deviation of an analog signal. A demand taken from our running example that incorporates all three of the aforementioned parameters might read as follows: *“A value failure of the output signal `a_set_fin` larger than 0,2V must be detected within 0.5ms (ASIL C)”*.¹⁸

In the following subsections, we will introduce the different parameter types provided by the VerSal language. There are primitive parameter types and there are composite parameters, which are composed of primitive parameters. Altogether there are the following parameters:

- Primitive Parameter Types: Boolean, Integer, Float and String
- Time
- Physical Quantities
- Error
- Integrity Level

Before the introduction of the different parameters, an explanation is needed as to how parameters are attached to language prototypes. Parameters are not directly attached to a language element by means of an attribute or a containment relation because we want to allow for conditional parameters. A conditional parameter is a parameter whose value depends on a condition (see section 4.3.2 for an introduction to conditions), for example a configuration condition.

¹⁸ This example demand is a “platform service failure detection demand” (see section 4.5.1) of the “analog output value failure” failure mode (see section 4.4.2.4).

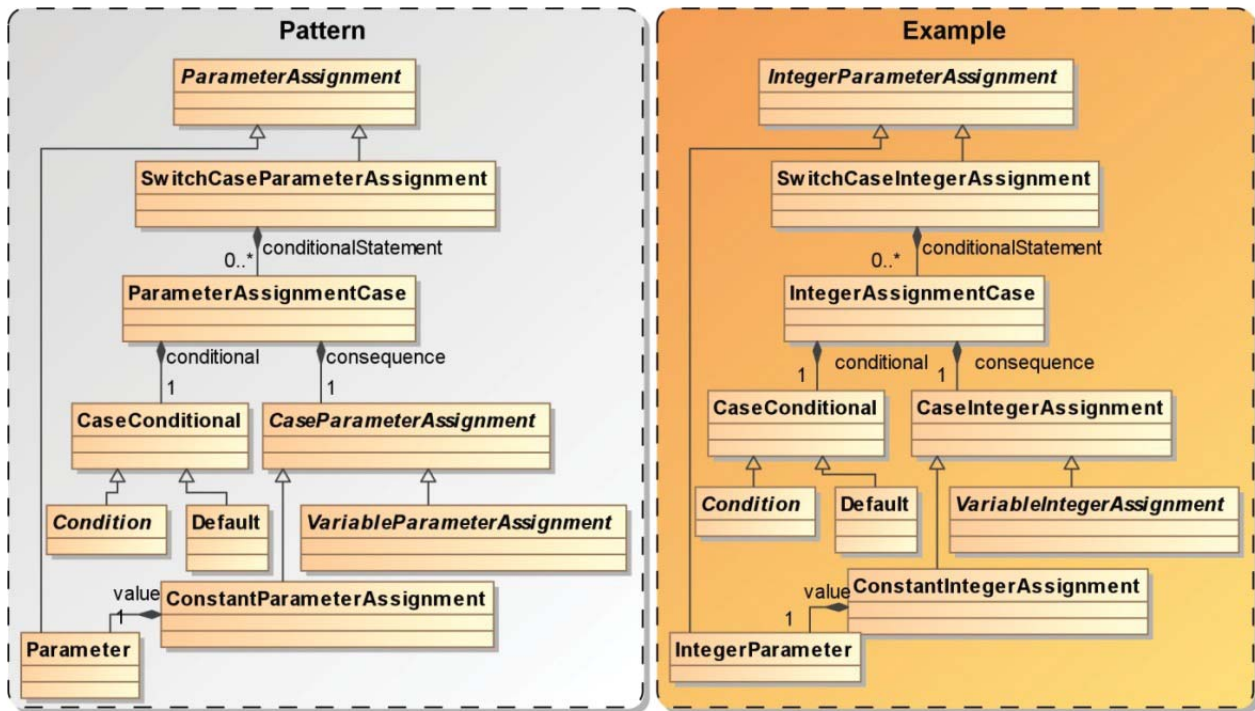


Figure 25: Parameter assignments : The left side of this figure shows the pattern for modeling parameter assignments. The right side shows an example parameter assignment for Integer parameters.

To realize conditional parameters, a parameterized interface requirement contains an abstract class called **ParameterAssignment**. There is one of these classes for every parameter (e.g., **IntegerAssignment**). Each **ParameterAssignment** class is inherited by the parameter itself, allowing direct unconditional parameter assignment, and by a class called **SwitchCaseParameterAssignment**. As the name of the class indicates, it allows for switch-case-like conditional parameter assignment. A switch assignment consists of a list of conditional/assignment tuples. The conditional part of such a tuple can be evaluated during mediation; if it is evaluated to true, the assignment part of the tuple describes which value should be assigned to the parameter. The evaluation of the switch parameter assignment is comparable to the evaluation of a switch statement in Java or C and is performed by the mediation algorithm. A more detailed explanation of the evaluation is provided in chapter 5. Figure 25 shows the modeling pattern of parameter assignments.

Primitive Parameter Types

The VerSal language contains four primitive parameter types: Integer, Boolean, String, and Float. We chose these types as they represent a sufficiently expressive starting point for modeling parameters. All primitive parameters are implemented as simple wrappers of the corresponding data types of the programming language used to

represent the meta-model (in our case this is Java). Figure 26 shows the implementation of the primitive parameter types in the VerSal language.

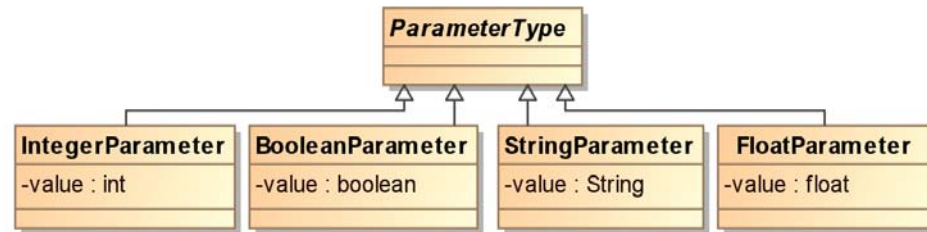


Figure 26: The primitive data types of the VerSal language

We used these primitive parameters to create standard composite parameters, which will be introduced in the following.

Time

A property that is often required to specify safety requirements is time. Time is needed for specifying parameters such as the tolerable jitters or latencies in signal transmission, or the tolerable detection intervals of specific failures. In VerSal, time is always modeled as an interval, i.e., the time that has elapsed between the occurrences of two events. In VerSal, there is no need to specify absolute points in time, like “2pm CET, August 8th, 2012”.

In VerSal, the time parameter is a composite parameter that is composed of two integer parameters. One integer defines the **milliseconds** and the other integer defines the **microseconds** of the time interval. The microseconds variable is not allowed to exceed the value of 999 in order to preserve the parameter’s canonical form. Furthermore, both integer variables have to be non-negative. Figure 27 shows the modeling of Time as a composite parameter.

In addition to the Time parameter, the VerSal language contains three auxiliary classes that support modeling interface requirements that require time parameters. These abstract classes are called **LatencyConstrained-InterfaceRequirement**, **TimeDeviationConstrainedInterface-Requirement**, and **JitterConstrainedInterfaceRequirement**. Each of these classes represents a common use case of time parameters in the specification of safety requirements. By inheriting from one of these classes, an interface requirement inherits time parameters that already have a fixed semantics.

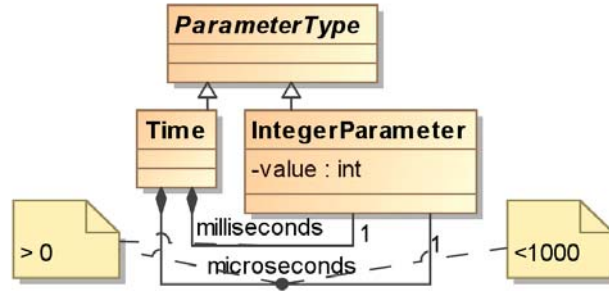


Figure 27: The meta-model of the composite time parameter

The *latency constraint* and the *time deviation constraint* are used to model interface requirements that depend on the specification of intervals. An example requirement is the demand for detecting a communication latency failure, which requires the specification of the nominal communication latency in order to distinguish nominal latencies from erroneous latencies. The latency constraint allows modeling the acceptable time interval by specifying the acceptable lower bound (t_1) of the latency, in case an early failure¹⁹ is critical, or the acceptable upper bound (t_u) of the latency, in case a late failure is critical. In case the early and the late failures are both critical, the corresponding acceptable time interval t_a , therefore, is $t_1 < t_a < t_u$. The time deviation constraint, on the other hand, defines the acceptable latency as a deviation (t_d) from the nominal latency (t_n). Consequently, the acceptable time interval t_a is $(t_n - t_d) < t_a < (t_n + t_d)$.

Unlike the latency and the time deviation constraints, the *jitter constraint* does not model a constraint regarding an arbitrary time interval, but rather the interval between the occurrences of two subsequent instances of the same periodical event. An example of such an event is the event that triggers the periodical sampling of an ADC, the receive event of a periodical message, or the scheduling of a periodical task. A period is defined by its duration (t_n), i.e., the nominal time between the occurrence of two subsequent instances of the same event, and its jitter (t_j), i.e., the admissible deviation from the nominal duration. The acceptable period p_a between two occurrences of the periodical event is $(t_n - t_j) < p_a < (t_n + t_j)$.

When an interface requirement inherits from one of the abstract time constraint classes, it inherits all its time parameters and the semantics that have just been specified. The events that specify the interval (i.e., the send event and the receive event of a message transmission define the transmission latency) are specific for each requirement. They can therefore not be inherited and have to be specified separately for each

¹⁹ The definition of late and early failure is provided in section 4.4.1.2.

requirement. Figure 28 shows the meta-model that specifies the time constraint classes introduced above.

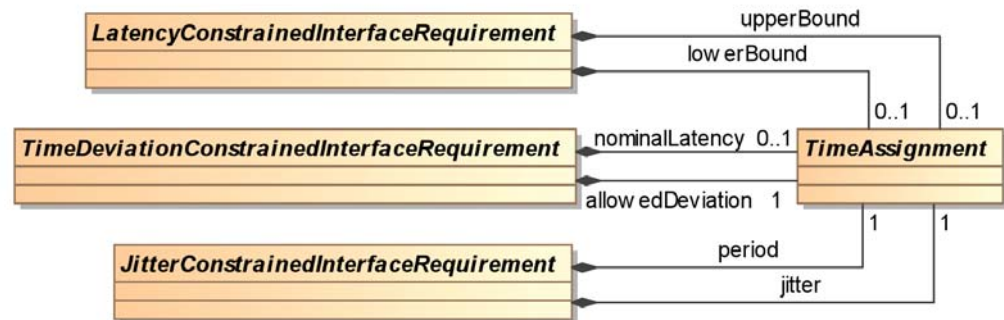


Figure 28: The meta-model of the time constraint classes

It is important to note that the design of the time-related parameters is based on existing model-based designs. The AUTOSAR standard, for example, contains a document describing the specification of time-related parameters in the AUTOSAR context. The “AUTOSAR Specification of the Timing Extension” [70] describes the specification of period and latency constraints in a way comparable to the VerSal language, together with many other AUTOSAR-specific timing constraints.

Physical Quantities

According to [71], a physical quantity describes a property of a phenomenon, body, or substance, where the property has a magnitude that can be expressed as a number and a unit of measurement. In the VerSal language, a physical quantity is used to specify physical signals at the interface between platform and application that have typically been read or written via analog input or output channels. In this use case, the relevant units typically are voltage, current, and frequency. However, the VerSal language does not restrict the user in the choice of units of measurement.

This is mainly because the current version of the VerSal language has no unit system. A unit is expressed by a string and the VerSal language only allows comparing two physical quantities if they have the same unit, i.e., if the strings of quantities are equal. In case the strings are not equal, an exception is thrown and the algorithm that issued the comparison has to react appropriately²⁰.

The number or value of the physical quantity is expressed by a floating point number in the VerSal language. Consequently, a physical quantity

²⁰ In our case, an appropriate reaction is the reaction that is pessimistic regarding mediation, i.e., makes mediation harder.

parameter is a composite parameter composed of a string parameter and a float parameter. The resulting meta-model is shown in Figure 29.

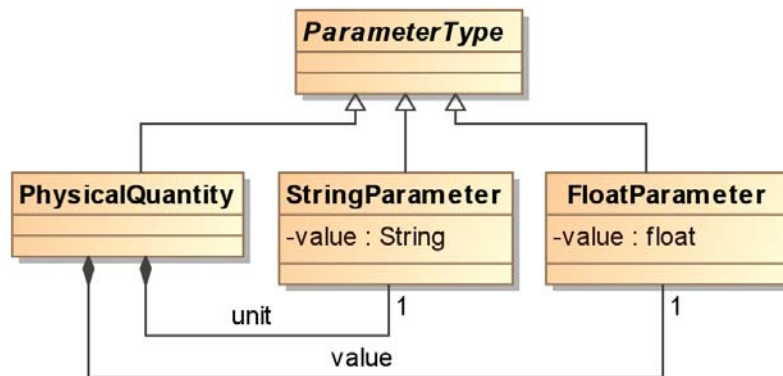


Figure 29: The meta-model of the physical quantity parameter

Error

An error parameter describes the admissible deviation of an actual value of a signal from the nominal value of the signal. An example usage of this parameter is the specification of a value failure of an analog input channel, where the error is used to specify the admissible deviation from the actual value returned by the ADC from the nominal value on the analog line. Please note that in a different context, the term error is also defined as the erroneous state of a component as a result of an internal fault or a failure of an interacting component.

The VerSal language allows the definition of absolute errors and relative errors. An absolute error is specified as a physical quantity that represents the absolute deviation of the actual value from the nominal value (e.g., an error of 0.2V). A relative error, on the other hand, describes the error from the nominal value relative to the quantity of the nominal value in percent (e.g., 10%).

Consequently, the error parameter is modeled as an abstract parameter with two sub-classes: the relative error and the absolute error parameter. Both classes are modeled as composite parameters: The relative parameter contains an integer parameter that must not be greater than 100, and the absolute parameter contains a physical quantity parameter. The corresponding excerpt of the VerSal meta-model is shown in Figure 30.

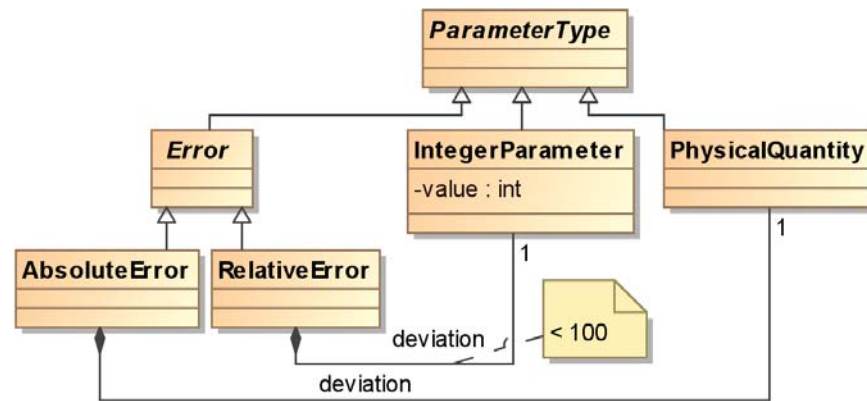


Figure 30: The meta-model of the composite error parameter

Integrity Level

The development of a safety-critical system is based on the concept of risk. The risk of a hazard is typically defined as the “product” of its probability and its severity. The go-to strategy for reducing the risk to an acceptable level includes reduction of the hazard’s probability of occurrence. The incorrect behavior of the system, i.e., failures, account for a certain part of the occurrence probability and must therefore be reduced. This is done by adding safety measures and designing the system in such a way that critical failures are avoided, contained, or detected and mitigated. The focus of our work is on specifying these design dependencies between applications and platforms and assessing whether a specific measure setup is adequate. However, the adequacy of a certain measure setup depends on the criticality of the failure that is protected by the measures.

The failures of a system are typically divided into systematic and random failures. Systematic failures are caused by design flaws, whereas random failures are typically caused by physical effects, like wear-out. Systematic failures are typically addressed by rigorous development processes and random failures by constraints regarding their occurrence probability.

As already stated in the related work chapter (see section 2.3), there are established methods like Component Fault Trees (CFTs) [55] for specifying failure logic modularly and for calculating failure rates from these modular specification. Therefore, our solution does not include another modular failure logic specification to support the specification of random failure rates. Instead, we suggest using an established method like CFTs.

Regarding systematic failures, most safety standards across most industrial domains regulate the rigor of their safety-related development processes using so-called integrity levels. The higher the required level of risk reduction, the higher the integrity level, and consequently, the more

rigorous the development processes. Typical integrity level scales are the safety integrity level (SIL) scale from IEC 61508 [59], the automotive safety integrity level (ASIL) scale from ISO 26262 [46], or the development assurance level (DAL) scale from DO-178C [60]. The development process also regulates the generation of process- and product-related evidences, which were introduced in section 2.3.

In accordance with the above-mentioned standards, each requirement specified using the VerSal language includes an integrity level specification. In case of a demand, this means that failing to fulfill the demand can lead to a hazard with the corresponding criticality. In case of a guarantee, this means that the guarantee has been developed according to the development process required for achieving the specified integrity level. Typically, guarantees used to fulfill a demand must be developed according to at least the same integrity level as the demand²¹.

Figure 31 shows the integrity level parameters that are represented in the meta-model. The part of the model describing integrity levels must be adapted when using different standards, since different standards use different integrity level scales. The figure shows a variant of the model that was adapted for use in the context of the automotive domain and the safety standard ISO 26262.

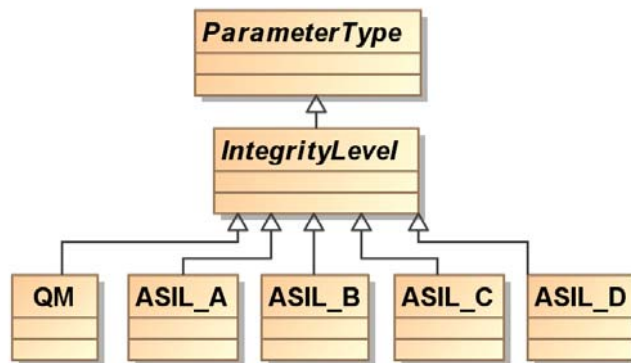


Figure 31: The meta-model of the integrity level parameter

The user of the VerSal method must attach an integrity level demand to every demand or guarantee, which is also the only use case of the integrity level parameter in the VerSal language. Figure 32 depicts the corresponding excerpt of the VerSal meta-model that shows the assignment of the integrity level parameter.

²¹ Some safety standards allow reducing the required integrity level by so-called decompositions, which is supported by the VerSal language.

4.4 Common Language – Failures and Failure Reactions

Product-related safety engineering often focuses on failure modes and safety mechanisms. Failure modes describe different ways in which the system under development fails, whereas safety measures are techniques and mechanisms that allow the system to control and tolerate these failures. Consequently, to reach our goal of providing a language that allows the specification of safety-related demands and guarantees between applications and demands, we have to provide a way for the specification of failure modes and safety measures. One of the core ideas behind the VerSal approach is to use the standardized services provided by an open integrated system to derive a standardized failure model and to use the most common safety measures to provide a standardized catalogue of failure reactions provided by an execution platform.

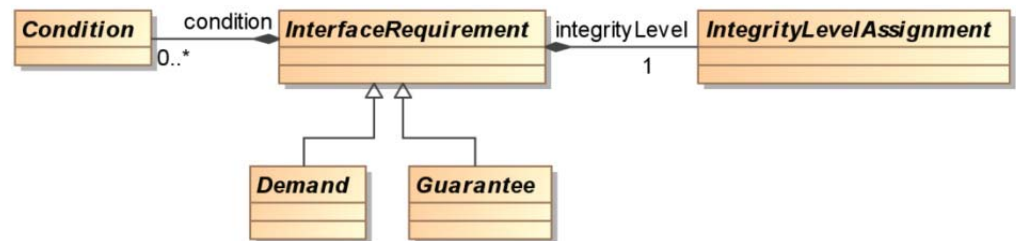


Figure 32: The assignment of the integrity level parameter to the top-level interface requirement class.

In this second part of the common language description, we introduce the standardized failure models and the failure reaction catalogue that can be used for the specification of the interface requirements in the VerSal language. In the upcoming sections 4.5 and 4.6, we will see that failure modes and failure reactions are used like types in the specification of demands and guarantees. We will also find out in chapter 5 that the standardization of the failure model and reaction types plays an essential role in allowing the mediation algorithm to automatically check whether a particular guarantee is suitable for fulfilling a given demand.

Since the failure model plays such a central role in the VerSal language, we dedicate section 4.4.1 to the description of the approach taken to analyze and specify the failure model. We will then continue with the introduction of our failure model, which differentiates between two kinds of common failure modes. Platform service failure modes describe failures of services that are offered by the platform, and will be introduced in section 4.4.2. On the other hand, application failure modes describe failures of the application that can be detected by a potential monitoring facility provided by the platform, and will be introduced in section 4.4.3. Section 4.4.4 finally describes the standardized platform failure reaction model.

4.4.1 Failure Analysis and Classification

A core part of the VerSal language is the common failure model that is used to specify demands and guarantees. We created the common failure model by analyzing the common platform services introduced in section 2.1.4 for failure modes. We performed the failure analysis using the guideword-based analysis based on the methods proposed by [64] and [72]. All things considered, both approaches introduce the same guidewords. These are:

- Service Value:
- Coarse Incorrect
- Subtle Incorrect
- Service Timing:
- Early
- Late
- Service Provision:
- Omission
- Commission

Together the guidewords form the failure mode topology shown in Figure 33.

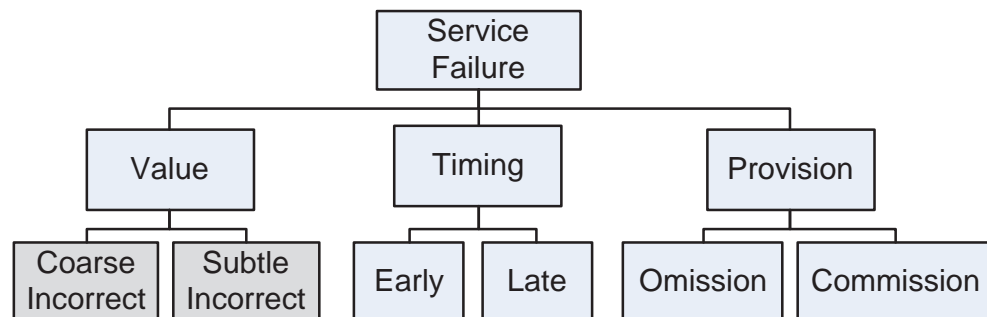


Figure 33: The failure mode taxonomy used in VerSal . Classes depicted by gray rectangles are not used in the language.

In the following sections, we will introduce the different failure classes: value, timing, and provision. In the corresponding sections, we will explain their original meaning as well as their interpretation and their usage in the VerSal language.

Service Value

Most services will eventually produce a value when invoked. However, it is also possible that a service invocation only leads to a change of the service's internal state and produces no output. Furthermore, there are services that do not only produce outputs when invoked but produce

outputs, for example, at specific times. Whenever a service produces an output that is not in the set of acceptable outputs for the current situation, a value failure occurs.

Both [64] and [72] differentiate between coarsely incorrect and subtly incorrect. A coarse incorrect value failure can be detected by the user, whereas a subtle incorrect failure cannot. This differentiation is irrelevant for the VerSal language. Whether the application is able to detect a failure is only implicitly visible at the safety interface. If the application was able to detect the failure, there would usually not be a detection demand for the platform. Whether the platform is able to detect a failure is clearly specified by the availability of a corresponding guarantee. If there is no guarantee, the failure is not detectable. Accordingly, there is no need for the differentiation between coarse incorrect and subtle incorrect in the context of the VerSal language.

To specify a value failure, the user of the VerSal language must be able to distinguish acceptable from unacceptable outputs, and nominal outputs from erroneous outputs. Consequently, every value failure mode in the common language allows the user to make this distinction, usually using the error parameter.

Service Timing

In the majority of safety-critical embedded systems, the timing of the service is equally important as its correct value. When the correct output is delivered either too early or too late, the value might be as hazardous as or even more hazardous than an incorrect value at the correct time.

Since the services provided by the platform are usually invoked by the application, most timing failures can be specified using the time interval between the invocation of the service and the time when the service produces its output or performs its reaction. If this interval is shorter than allowed, there is an early timing failure, and if this interval is longer than allowed, there is a late timing failure.

Thus, in order to specify a timing failure, each failure has to include a specification of the start event and the end event of the corresponding interval, and the language user has to specify the timing thresholds that distinguish an admissible timing from an erroneous timing. This is always done using the abstract time constraints introduced in section 4.3.4 Time.

Please note that the language does not provide a late and early failure mode for every possible API call. Sometimes the API call is only part of a larger functionality provided by the service; in this case, the timing failure is specified on the service level. Furthermore, if the functionality is performed by one API call and that particular call is uninterrupted (there

is no wait involved) and synchronous, there will also be no timing failure. The timing of such an API call that behaves like a regular function call is already covered by the execution time of the calling ASWC.

In addition to late and early failures, we allow specifying so-called jitter failures. A jitter is the deviation from the periodicity of a periodical event. A jitter could also be replaced by a late and an early failure, but for reasons of convenience, jitter failures are directly specifiable using the VerSal language.

Service Provision

Service provision is differentiated into omission and commission failure modes. An omission failure occurs if the service produces no output even though an output should have been produced. A commission failure occurs if the service produces an output event though the output should not have been produced. However, there are some difficulties regarding omission failures.

First and foremost, neither the user nor the system can differentiate an omission failure from an infinitely late failure. If the service has not produced any output after a given time, the system has no chance of knowing whether the service will eventually produce an output. In such a case, the user has to specify a certain time interval, after which a service omission is assumed. However, if the service is supposed to eventually produce an output, the system might no longer be able to associate the output with the correct invocation and treat the output as a commission.

Furthermore, it is sometimes hard to differentiate an omission failure from a value failure if there is no NIL representation in the service's output domain. As an example, let us examine an analog output channel that produces a voltage signal. There is always a potential on the output channel. Let us assume the actual potential is v_{current} . Let us further assume that the application demands outputting a new potential v_{new} . If the output channel still produces v_{current} after the time interval assigned to an omission, there is no way of telling a value failure that produced v_{current} from an omission failure. Consequently, we decided not to use omission failures in case the analyzed service has no inherent NIL representation.

Since a commission failure is specified as producing an output even though the service should be inactive, in the VerSal language a commission failure is specified by the output or the reaction that should not have been performed and the condition that nominally inhibits the service. The specification of a commission failure follows the template below:

Example: **Service X performs action Y even though condition A or condition B or ... or condition C were true.**

Since an omission failure is specified as not producing an output even though the service should produce an output, in the VerSal language an omission failure is specified by the output or the reaction that should have been performed and the conditions that nominally activate the service. The specification of an omission failure follows the template below:

Example: **Service X does not perform action Y even though condition A or condition B or ... or condition C were true.**

4.4.2 Platform Service Failures

The platform service failures package contains a standardized set of configurable failure models of typical platform services. Each service-specific failure model contains a set of failure modes that are commonly used for both the specification of application demands and platform guarantees. The VerSal language supports failure models for the platform services identified in section 2.1.4. These are:

- Synchronization Services
- Communication Services
- Input Services
- Output Services
- Time Services
- Memory Services
- Scheduling
- Basic Execution Services

In the following subsections, we will present a failure model for each service class. Each failure model contains a set of failure modes describing different ways of how the analyzed service can potentially fail.

Subsequently, we will introduce the failure models of the different service classes. Each failure model specification starts with a description of the corresponding service class including a description of the functionality provided and the different use-case scenarios of the service class. After the introduction of the service class we will illustrate the different failure modes of the service class including parameters (see section 4.3.4 for more information on parameters) and architecture relations (see section 4.3.3 for more information on architecture relations).

Please note that every failure mode in the platform service failure model is a failure mode prototype or, in other words, a failure mode class, rather than a specific failure mode. In order to turn the failure mode prototype into a failure mode, it has to be “instantiated”. With instantiation we mean that the failure mode has to be parameterized and related to an architecture element. The relation to an architecture element turns a failure mode type into a failure mode of that specific architecture element. If, for instance, we relate a common communication corruption failure to the **v_ref** communication port of our running example, the failure mode type is instantiated as a failure mode of the specific signal received via the **v_ref** port.

As a final remark before the introduction of the failure models, we note that the VerSal language does not allow specifying demands or guarantees regarding 2nd level failures (failures of safety measures); therefore, there are no failure modes of platform failure detection or platform failure reaction mechanisms.

Synchronization Failure Model

The synchronization failure model describes the failure modes of a platform’s synchronization mechanisms as introduced in section 2.1.4. A synchronization mechanism allows the application developer to control the execution sequence or, in other words, the control flow between several runnables²². To do so, a synchronization mechanism contains at least one so-called blocking call, which sets the task²³ executing the runnable to the **waiting** state. Instead of blocking calls provided, for example, by the time services, the waiting task is not released to the **ready** state by the operating system (at least in the nominal case) but by another runnable.

There can be numerous types and implementations of synchronization mechanisms in a modern operating system. We chose to cover simple implementations of the two most common synchronization mechanisms with the VerSal language. Other mechanisms and more complex implementations are left open for future extensions. The failure model covers an abstract mutual exclusion (mutex) mechanism for implementing critical regions and an event mechanism that allows signaling between runnables.

A critical region refers to a sequence of instructions in which a program accesses a shared resource (like a shared variable or a system register

²² A runnable is a schedulable entity of an ASWC. Refer to Annex A.2 for more information about the application model.

²³ A task is an atomic schedulable entity of the platform. A runnable runs in the context of a task. Refer to Annex A.3 for more information about the platform model.

controlling a device) in a non-atomic way. If the program is interrupted during this process and another program starts accessing the shared resource, the resource might reach an undefined or inconsistent state, jeopardizing the correct execution of both programs. To prevent this predicament from happening, all programs accessing the shared resource enter the afore-described critical region before they start to manipulate the resource. The mutex mechanism ensures that only one program is able to enter the same critical region at a time.

A mutex, as defined in the context of our language, contains two procedures, **enter mutex** to enter the critical region and **exit mutex** when the critical region is exited. Furthermore, the **enter mutex** procedure allows the application to specify a timeout in order to define the maximum amount of time the application wants to wait in case the mutex is occupied. When **enter mutex** is called, the call directly returns if the mutex is free and the call blocks the calling task if the mutex is occupied. The waiting task is put into the ready state as soon as the mutex is exited or the specified timeout has expired. In case the mutex has been exited, the activated task makes another attempt to enter the mutex. The described mutex mechanism is comparable to a binary semaphore (a semaphore with only two states) and so are the failure modes. However, failure modes of counting semaphores (semaphores that allow an arbitrary resource count) are not covered by our language. The mutex mechanism is illustrated in Figure 34.

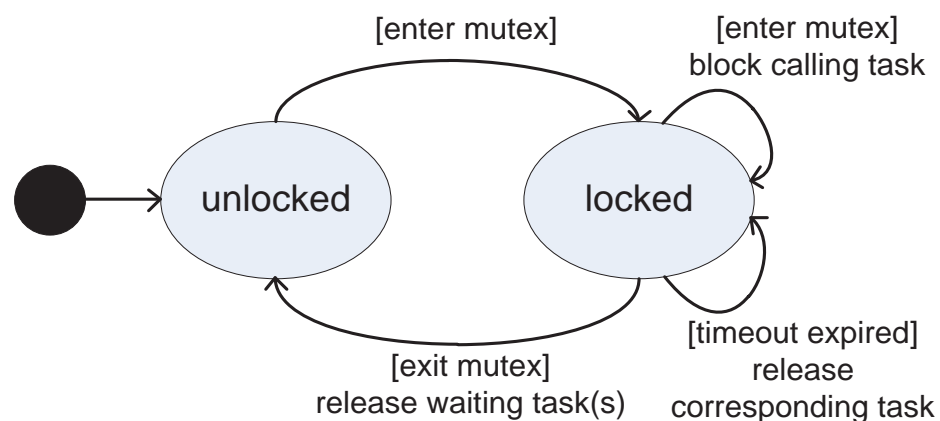


Figure 34: The state machine of a mutex

Contrary to the mutex mechanism, we have to differentiate two different user roles when describing the event mechanism. First, there are programs waiting for a particular event and second, there are programs signaling the event. When a runnable calls **wait event**, the corresponding task enters the **waiting** state (events are not stored). As soon as another runnable calls **signal event** for the corresponding event, all tasks waiting for the event are released into the **ready** state. Comparable to the timeout feature of the mutex mechanisms, the event mechanism contains a timeout feature as well.

We will now continue describing the failure modes of the synchronization failure model containing six different failure modes. An overview of the synchronization failure mode is given in Figure 35. The failure model consists of the following failure modes:

- synchFM-1: Mutex Access Commission
- synchFM-2: Mutex Access Omission
- synchFM-3: Mutex Release Commission
- synchFM-4: Mutex Release Omission
- synchFM-5: Mutex Timeout Failure
- synchFM-6: Event Signal Commission
- synchFM-7: Event Signal Omission
- synchFM-8: Event Timeout Failure

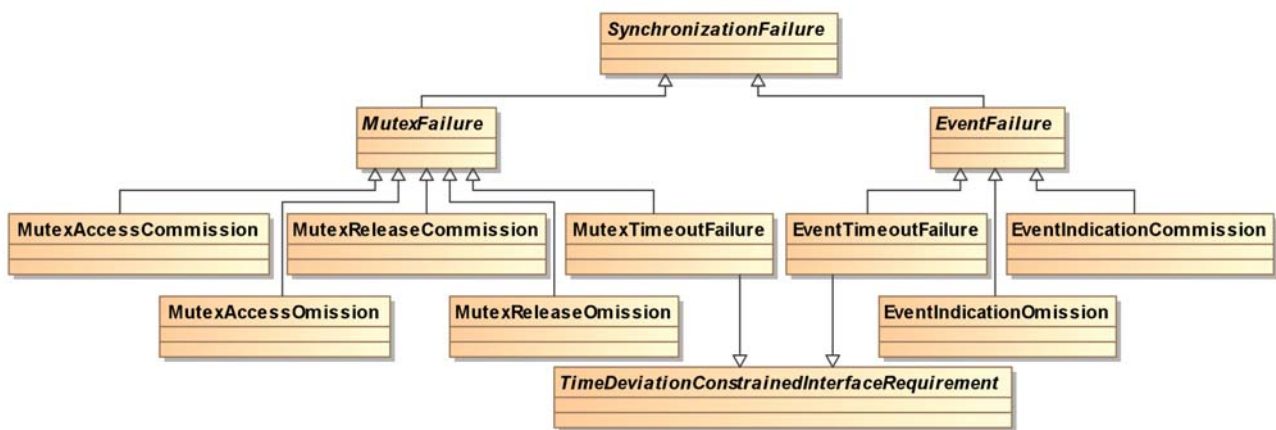


Figure 35: The meta-model of the synchronization failure model

synchFM-1: Mutex Access Commission

A mutex access commission occurs if a runnable calling `enter mutex` is allowed to enter the critical region even though the region is occupied.

synchFM-2: Mutex Access Omission

A mutex access omission occurs if a runnable calling `enter mutex` is blocked even though the region is not occupied.

synchFM-3: Mutex Release Commission

A mutex release commission occurs if a runnable is released even though the region is not left.

Please note that this failure does not implicate that the released runnable is allowed to enter the mutex even though the mutex is still occupied.

synchFM-4: Mutex Release Omission

A mutex release omission occurs if a runnable is not released even though the region is left.

synchFM-5: Mutex Timeout Failure

A mutex timeout failure occurs if a runnable that has been blocked for calling `enter mutex` with a timeout parameter is released too early before or too late after the timeout has expired.

In order to specify the tolerable deviation from the specified timeout failure, the event timeout failure is parameterized according to the time deviation constraint introduced in section 4.3.4 Time.

synchFM-6: Event Signal Commission

An event signal commission occurs if a runnable waiting for an event gets released even though no runnable has called a corresponding `signal event` operation.

synchFM-7: Event Signal Omission

An event signal omission occurs if a runnable waiting for an event gets not released even though a runnable has called a corresponding `signal event` operation.

synchFM-8: Event Timeout Failure

An event timeout failure occurs if a runnable that has been blocked for calling `wait event` with a timeout parameter is released too early before or too late after the timeout has expired.

In order to specify the tolerable deviation from the specified timeout failure, the event timeout failure is parameterized according to the time deviation constraint introduced in section 4.3.4 Time.

Communication Failure Model

This subsection describes the failure model of a platform's communication service as introduced in subsection 2.1.4. The communication failure model was derived from existing failure models introduced in common safety standards like [46] or [73].

In this context, communication means the exchange of information between ASWCs, or in other words, the exchange of logical signals between ASWCs. In such a communication scenario, the ASWC providing the information is called the sender ASWC and the ASWC

requiring the information is called the receiver ASWC. Since we examine communication on a functional level, the communication process is regarded as event-based rather than continuous, as we would have to regard it on a physical level. The entity that is transmitted during one communication instance is called a message; the payload of a message is called data. The event at the beginning of the communication process is called the **send event**, the event at the end of the communication process is called the **receive event**. The send event is triggered as soon as the data are available for communication at the PSW of the sender ASWC. The receive event is triggered as soon as the PSW of the receiver ASWC has made the data available to the receiver ASCE, which includes an indication if receive notification is enabled. The described scenario is depicted in Figure 36.

As discussed earlier, the deployment of applications is only determined after the development of the application. Consequently, the communication-related failure model must neither differentiate between inter-process, inter-partition, or inter-platform failure modes, nor between failure modes specific to a certain technical implementation like CAN or FlexRay. Therefore, the presented communication failure model contains all failures that can occur in an embedded systems communication scenario.

The communication failure model contains six basic failure modes; an overview of the failure model is shown in Figure 37. Each failure mode type can be configured as a provided or as a required message failure mode, so that sender ASWCs and receiver ASWCs alike are able to specify demands regarding communication. The basic failure model contains the following failure modes:

- comFM-1: Message Corruption
- comFM-2: Message Insertion
- comFM-3: Message Loss
- comFM-4: Incorrect Message Sequence
- comFM-5: Late Transmission
- comFM-6: Early Transmission

Prior to introducing the communication failure modes, two remarks are in order, one on the design and one on the interpretation of the communication failure model.

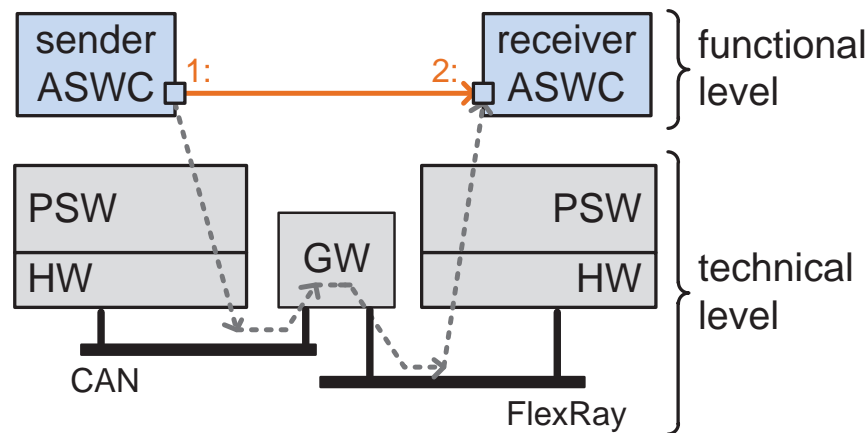


Figure 36: Comparison of functional and technical communication scenarios . Only the functional level is regarded in the application language. GW stands for gateway. The following events are shown in the figure. 1: the send event; 2: the receive event.

First, every demand, no matter whether it is specified on the sender or on the receiver side, is targeted at the complete communication path. When, for example, a *receiver* ASWC demands timely transmission of a message, this does not only call for timely behavior of the *receiving* platform, but of all other platforms involved in the communication process. This design decision was made because the alternative design, splitting responsibilities between sender and receiver, has two disadvantages. First, there would be an implicit dependency between the vertical interfaces of the sender and receiver ASWC, since neither the receiver demand nor the sender demand would be meaningful without the other. With such a dependency, the vertical interfaces would include a “hidden horizontal interface”, as there would not only be a dependency between one application and one platform but also between the involved (sender and receiver) applications. The second reason for not splitting responsibilities is the routing of a signal, which is potentially very complex. Which demand (the sender or the receiver one) would include the responsibility for the safe behavior of gateways?

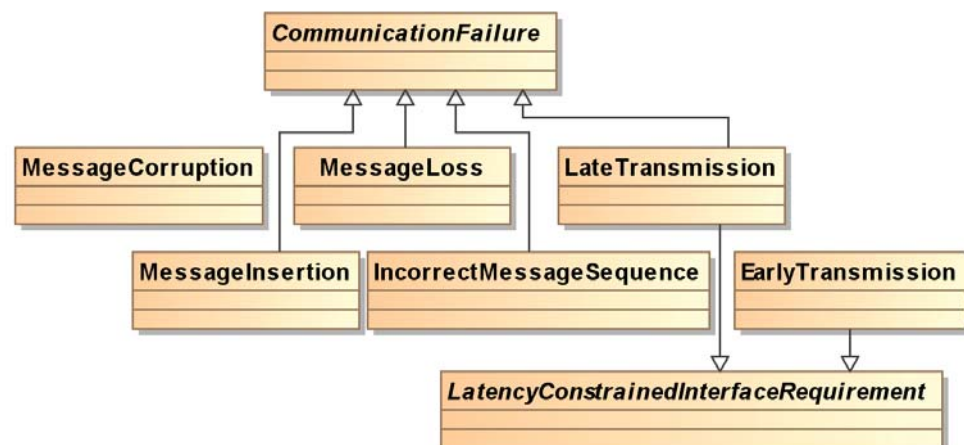


Figure 37: The meta-model of the communication failure model

Second, we would like to clarify that every communication failure mode is specified from the receiver's point of view. Every failure mode describes a deviation of a correct communication as perceived by the receiver. To exemplify this, let us regard two scenarios. In both scenarios the sender ASWC sends a message that gets transmitted faster than specified. In the first scenario, the message overwrites an earlier message at the port of the receiver ASWC before the ASWC has been able to read the message that is now overwritten. In the second scenario, the message reception triggers an early reaction of the receiver ASWC. Unlike the second scenario, the first scenario is not considered to be an early message failure. This is because the receiver perceives this failure mode either as a deleted or as a corrupted message. In this case, the fast or early transmission is a technical cause for a message corruption or a message loss failure. Technical causes for failures are not regarded on the level of the interface language.

comFM-1: Message Corruption

Message corruption occurs if the data received by the receiver ASWC are not identical to the data originally sent by the sender ASWC.

Since application demands are specified on a functional level, data would still be considered as identical if they are changed for technical reasons. This happens, for example, if receiver and sender use different endianness. Message corruption is a common communication failure mode, which may, for example, be caused by electro-magnetic influences.

comFM-2: Message Insertion

Message insertion occurs if the receiver ASWC receives a message that has not been sent by a valid sender ASWC.

Typically, one distinguishes two cases of this failure. In the first case, a valid sender²⁴ sends a superfluous message, or a valid message is accidentally duplicated on the communication link. In the second case, an unauthorized sender assumes the identity of a valid sender and sends a message on its behalf. The latter case is typically labeled as masquerading. Since one cannot distinguish between the two cases on the application level, there is no additional failure mode to differentiate masquerading in the application language. Both unintentional duplication and masquerading are regarded as a message insertion.

²⁴ A sender that is allowed to send this type of message according to its specification

comFM-3: Message Loss

Message loss occurs if the receiver ASWC does not receive a message that has been properly sent by a valid sender ASWC.

comFM-4: Incorrect Message Sequence

An incorrect message sequence occurs if two messages that have been sent successively by the sender ASWC are received out of order by the receiver ASWC.

The specification of the incorrect message sequence failure has two restrictions. First, it allows demanding the correct sequence of only two sequential messages. With this demand, the designer is able to pessimistically include failures in longer sequences, since any failure in a sequence of more than two messages implies a sequence failure of at least two messages. Second, the designer is only able to demand the correct sequence of messages sent or received via the same port.

comFM-5: Late Transmission

A late transmission failure occurs if the transmission latency is too large. Transmission latency is defined as the time interval between the **send event** and the **receive event**.

At the point in time when a late message failure is detected one cannot determine whether the message is really delayed or actually lost. A lost message would not be in time either. This means that every measure for detecting late messages is also effective for detecting lost messages. Yet on the other hand, there are measures for detecting lost messages that cannot detect late messages. Therefore, message loss is regarded separately and not as a special case of the late message failure.

In order to specify the tolerable maximum transmission latency, the late transmission failure is parameterized according to the latency constraint introduced in section 4.3.4 Time.

comFM-6: Early Transmission

An early transmission failure occurs if the transmission latency is too small. Transmission latency is defined as the time interval between the **send event** and the **receive event**.

In order to specify the tolerable minimum transmission latency, the early transmission failure is parameterized according to the latency constraint introduced in section 4.3.4 Time.

Input Failure Model

This subsection describes the failure model of the input part of a platform's input/output functionality as described in subsection 2.1.4. The input functionality allows an application to connect to sensors via the platform's input peripherals. We do not differentiate between on-chip peripherals that are directly accessible via the platform's processing unit (on-chip peripherals) and on-board peripherals that are connected via on-board communication busses like SPI or I²C.

The failure model does, however, differentiate between digital and analog input channels. Digital input channels are typically implemented by appropriately configured DIO (digital input/output) pins of a microcontroller to read the status of digital switches connected to the platform. DIO pins can also be used as a source for interrupt triggers, but this functionality is covered by the scheduling failure model. Analog input channels, on the other hand, are usually implemented using ADCs (Analog-to-Digital Converter).

Analog input channels, i.e., ADCs, and digital input channels, i.e., DIOs, are accessed differently. Since an ADC conversion takes many CPU cycles, they are typically accessed asynchronously, whereas DIOs are accessed synchronously (on modern microcontrollers with sophisticated memory architectures, DIO access takes several CPU cycles as well, but not nearly as many as ADC conversion). These different scenarios for accessing digital and analog input peripherals result in different failure modes, which is why the input failure model differentiates between digital and analog input failure modes.

To elaborate on the failure model, we will first describe the different input scenarios. We divide a typical input scenario into the data acquisition phase and the data access phase. The data acquisition phase begins with the ASWC requesting the input peripheral to read from the physical input channel and ends when the input peripheral starts to read from the physical input channel. The data access phase, on the other hand, starts when the input peripheral has finished reading from the physical input channel and ends when the acquired data is provided to the respective ASWC. An overview of the different scenarios is shown in Figure 38.

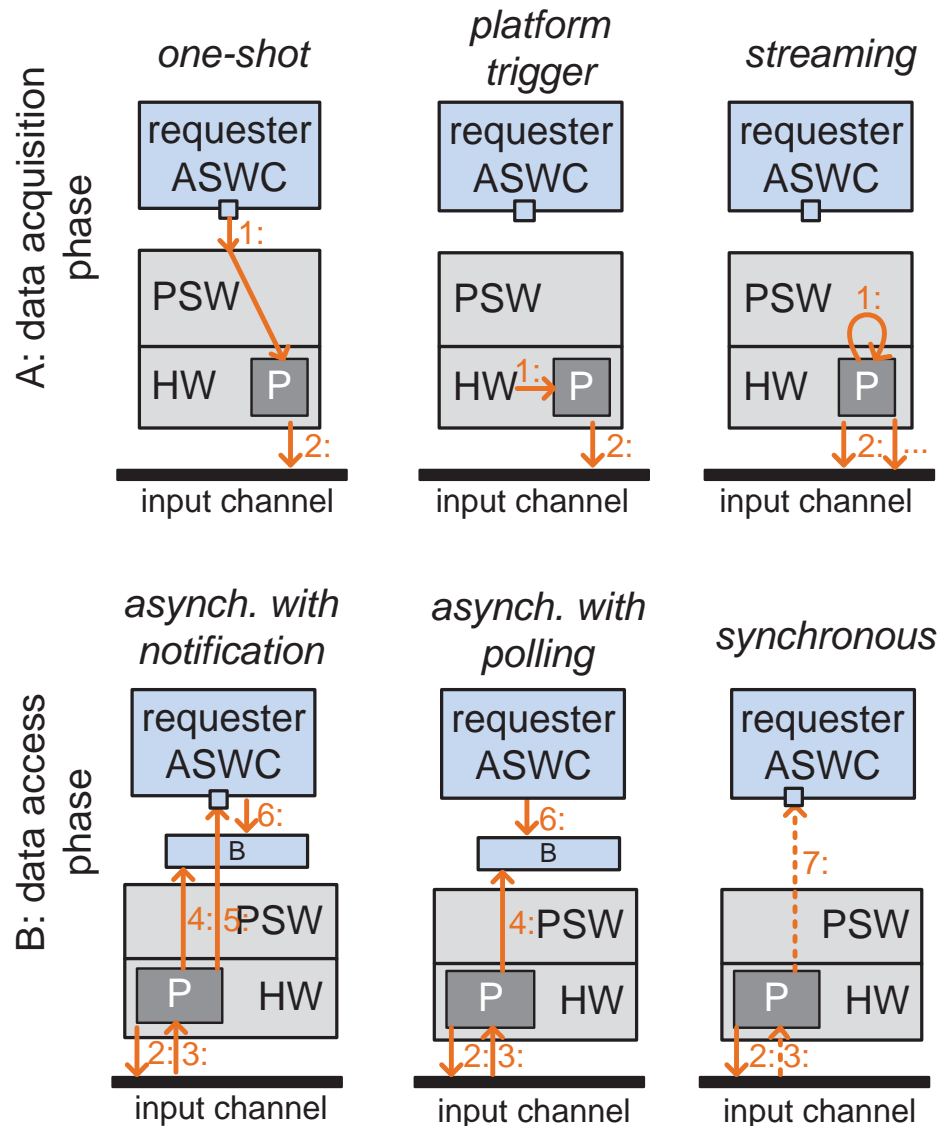


Figure 38:

The different scenarios for accessing an input channel . The following events are shown in the figure: 1: read/sampling requested; 2: read/sampling started; 3: read/sampling finished; 4: result copied; 5: notification sent; 6: result read; 7: returned with result.

There are three different kinds of read/sampling requests. When working with a standard ADC, one has to differentiate between the so-called one-shot mode, externally triggered sampling (in our case called platform trigger), and streaming mode. In the first scenario, the application triggers the ADC to sample the input channel one time by calling the platform API. In the second scenario, the ADC is triggered by the platform (for example by a timer), which is typically used to implement periodical sampling. In the last scenario, the ADC runs in the so-called streaming mode, where the ADC starts a new sampling procedure as soon as the last one has ended. When reading from a DIO channel, data acquisition is typically only triggered by the application in a one-shot manner.

On the other hand, the data access phase begins with the completion of the sampling procedure. Since the ADC sampling takes several CPU cycles, samples are typically accessed asynchronously. When the sampling is completed, the platform software copies the data into a data buffer located in a memory area accessible to the ASWC (this memory area could be a part of the ASWC's private memory). After this common step, we differentiate between asynchronous data access with and without notification. In the first scenario, the platform notifies the ASWC when the sampling has been completed and the data have been copied, whereas in the second scenario, the ASWC is left without notification and has to poll the buffer or the PSW for status information. On the other hand, reading from a DIO channel is typically done synchronously, which might, however, involve a trap and a switch to supervisor mode.

As we will explain later, certain failure modes of the input failure model are interpreted differently or do not apply depending on the input scenario relevant for the current input channel. In sum, the input failure model contains fifteen different failure modes. The analog input failures of the input failure model are depicted in Figure 39; the digital input failure are modeled accordingly. The failure modes of the input failure model are:

- inFM-1: Digital Input Read Omission
- inFM-2: Digital Input Late Read
- inFM-3: Digital Input Early Read
- inFM-4: Digital Input Late Return
- inFM-5: Digital Input Early Return
- inFM-6: Digital Input False Positive
- inFM-7: Digital Input False Negative
- inFM-8: Analog Input Omission
- inFM-9: Analog Input Commission
- inFM-10: Analog Input Late Sampling
- inFM-11: Analog Input Early Sampling
- inFM-12: Analog Input Sampling Jitter
- inFM-13: Analog Input Late Return
- inFM-14: Analog Input Early Return
- inFM-15: Analog Input Value Failure

Reading from a digital input is usually performed in a synchronous way. As described in section 4.4.1 Service Timingsection, we usually do not specify timing failure modes for those services. However, if a digital input channel is read by a complex driver, the driver might perform the writing asynchronously. Because of these cases, we have added timing failures to the digital input failure model.

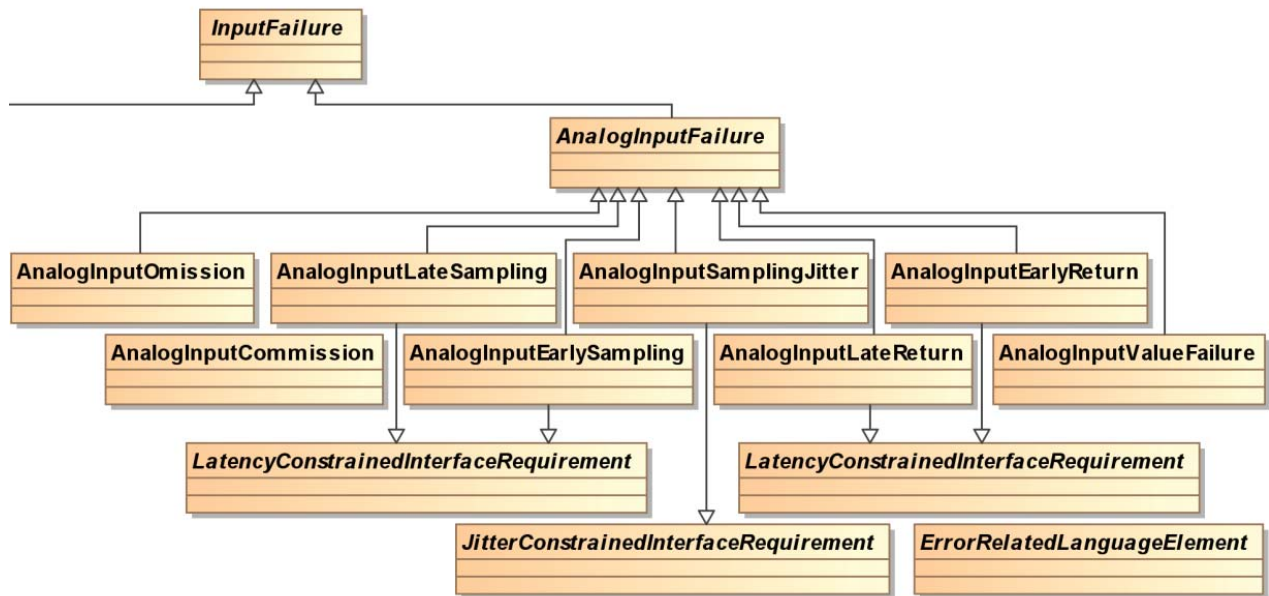


Figure 39: The meta-model of the analog part of the input failure model

inFM-1: Digital Input Read Omission

A digital input omission occurs if the read request does not return or returns with an error value.

If the digital read is implemented in a truly synchronous way, the request will always return if no exception occurs. If the read request returns with an error code, this is considered as a detected read omission.

inFM-2: Digital Input Late Read

A digital input early read failure occurs if the delay between the **read requested** event and the **read started** event is larger than the tolerable maximum delay.

The delay between the read request and the start of the reading procedure is relevant for determining the age of the data when the data is returned to the requester ASWC.

In order to specify the tolerable maximum read delay, the digital input late read failure is parameterized according to the latency constraint introduced in section 4.3.4 Time.

inFM-3: Digital Input Early Read

A digital input early read failure occurs if the delay between the **read requested** event and the **read started** event is smaller than the tolerable minimum delay.

Usually, we assume that an early read is uncritical. Nevertheless, for reasons of completeness, we added the early read failure mode to the input failure model.

In order to specify the tolerable minimum read delay, the digital input early read failure is parameterized according to the latency constraint introduced in section 4.3.4 Time.

inFM-4: Digital Input Late Return

A digital input late return failure occurs if the end-to-end delay between the request of the read procedure (**read requested** event) and the return of the read procedure (**returned with result** event) is too large.

The return delay is important for determining the end-to-end delay (from sensor to actuator) of the corresponding application.

In order to specify the tolerable maximum return delay, the digital input late return failure is parameterized according to the latency constraint introduced in section 4.3.4 Time.

inFM-5: Digital Input Early Return

A digital input late return failure occurs if the end-to-end delay between the request of the read procedure (**read requested** event) and the return of the read procedure (**returned with result** event) is too small.

In order to specify the tolerable minimum return delay, the digital input early return failure is parameterized according to the latency constraint introduced in section 4.3.4 Time.

inFM-6: Digital Input False Positive

A digital input false positive failure occurs if the Boolean value returned to the requester ASWC is **true**, even though the value on the digital input channel was **false** at the time the read process started.

inFM-7: Digital Input False Negative

A digital input false negative failure occurs if the Boolean value returned to the requester ASWC is **false**, even though the value on the digital input channel was **true** at the time the read process started.

inFM-8: Analog Input Omission

The semantics of the analog input omission failure depends on the **triggerType** and the **returnType** parameter of the **AnalogSensorInPort** or the **AnalogInputChannel** related to the failure mode.

In case the ADC is triggered by the platform (**triggerType** = **PlatformTrigger**) or operates in streaming mode (**triggerType** = **Streaming**), an omission occurs if there is no trigger (**sampling request** event) even though there should be a trigger regarding the specification of the platform. In those scenarios, triggering the sampling is not the responsibility of the ASWC.

Furthermore, a sampling omission occurs if the sampling has been triggered but the platform omits copying the sampled value into the respective result buffer of the requester ASWCs. In case notifications are enabled (**returnType** = **Notification**), an omission also occurs if the platform copies the sampled value into the buffer but omits to notify the ASWC.

inFM-9: Analog Input Commission

The semantics of the analog input omission failure depends on the **returnType** parameter of the **AnalogSensorInPort** or the **AnalogInputChannel** related to the failure mode.

If notifications are disabled (**returnType** = **Polling**) an analog input commission occurs if the platform copies a new value into the data buffer even though there was no ASWC trigger or there should not have been a trigger (in case trigger responsibility was with the platform). If notifications are enabled (**returnType** = **Polling**), an analog input commission occurs if the platform sends a notification even though there was no ASWC trigger or there should not have been a trigger (in case trigger responsibility was with the platform).

inFM-10: Analog Input Late Sampling

An analog input late sampling occurs if the delay between the **sampling requested** event and the **sampling started** event is too large.

The delay between the sampling request and the start of the sampling procedure is relevant for determining the age of the data when the data is returned to the requester ASWC.

The analog input late sampling failure mode only applies if the ADC is triggered by the ASWC (**triggerType** = **OneShot**) or by the platform

(**triggerType** = **PlatformTrigger**), since there is no delay between the trigger and the beginning of the sampling in streaming mode (**triggerType** = **Streaming**).

In order to specify the tolerable maximum sampling delay, the analog input late sampling failure is parameterized according to the latency constraint introduced in section 4.3.4 Time.

inFM-11: Analog Input Early Sampling

An analog input early sampling occurs if the delay between the **sampling requested** event and the **sampling started** event is too small.


The delay between the sampling request and the start of the sampling procedure is relevant for determining the age of the data when the data is returned to the requester ASWC.

The analog input late sampling failure mode only applies if the ADC is triggered by the ASWC (**triggerType** = **OneShot**) or by the platform (**triggerType** = **PlatformTrigger**), since there is always no delay between the trigger and the beginning of the sampling in streaming mode (**triggerType** = **Streaming**).

In order to specify the tolerable minimum sampling delay, the analog input early sampling failure is parameterized according to the latency constraint introduced in section 4.3.4 Time.

inFM-12: Analog Input Sampling Jitter

An analog input sampling jitter failure occurs if the time between two successive periodical sampling triggers (**sampling requested** event) varies too much.

Such a variation has detrimental effects on the precision and the stability of a closed loop control algorithm. To make a demand regarding the end-to-end jitter of a periodical sampling procedure, the analog input sampling jitter demand must be combined with a return latency failure (see )

The analog input sampling jitter failure only applies if the ADC is triggered by the platform (**triggerType** = **PlatformTrigger**), since in one-shot mode, the ASWC has the trigger responsibility and in streaming mode, periodical triggering is impossible.

In order to specify the tolerable jitter, the analog input sampling jitter failure is parameterized according to the jitter constraint introduced in section 4.3.4 Time.

inFM-13: Analog Input Late Return

The semantics of the analog input late return failure depends on the **returnType** parameter of the **AnalogSensorInPort**.

In case of a polling return type (**returnType** = **Polling**), an analog input late return occurs if the delay between the **sampling requested** event and the **result copied** event is too large.

In case of a notification return type (**returnType** = **Notification**), an analog input return latency failure occurs if the delay between the **sampling requested** event and the **notification send** event is too large.

In order to specify the tolerable maximum return delay, the analog input late return failure is parameterized according to the latency constraint introduced in section 4.3.4 Time.

inFM-14: Analog Input Early Return

The semantics of the analog input early return failure depends on the **returnType** parameter of the **AnalogSensorInPort**.

In case of a polling return type (**returnType** = **Polling**), an analog input late early occurs if the delay between the **sampling requested** event and the **result copied** event is too small.

In case of a notification return type (**returnType** = **Notification**), an analog input return latency failure occurs if the delay between the **sampling requested** event and the **notification send** event is too small.

In order to specify the tolerable minimum return delay, the analog input early return failure is parameterized according to the latency constraint introduced in section 4.3.4 Time.

inFM-15: Analog Input Value Failure

An analog input value failure occurs if the actual value on the analog channel at the **sampling started** event and the value returned to the requester ASWC deviate by more than the predefined error.

The error can be absolute (e.g., 0,5V) or relative (e.g., 10%). See section 4.3.4 Error for more information about modeling errors.

Output Failure Model

This subsection describes the failure model of the output part of a platform's input/output functionality as described in subsection 2.1.4. A platform's output functionality allows the application to access the platform's output peripherals in order to connect to the application's actuators. We assume that the application cannot differentiate between output channels implemented by on-chip and on-board peripherals, which is why the failure model in this subsection does not differentiate between the two cases either.

Unlike the input failure model, the output failure model does not differentiate between different scenarios of accessing an output peripheral, resulting in a less complex failure model. This is mainly because accessing an output peripheral does not include returning a value, which is why we do not have to differentiate between synchronous and an asynchronous access. The output scenario starts with the output request event when the ASWC requests the output peripheral to update the activation signal sent to the actuator, and ends when the output peripheral has written the new value to the output channel. The scenario for accessing an output peripheral is shown in Figure 40.

Because of the reasons described in section 4.4.1 Service Provision, we decided not to include an omission failure mode in our failure model since an omission failure (not processing an output request) cannot be differentiated from a value failure, as there is no **NIL** concept on a physical channel. It is sufficient to demand that the correct value has to appear on the channel after a certain period of time. Not processing the output request at all is just a technical cause of this kind of failure.

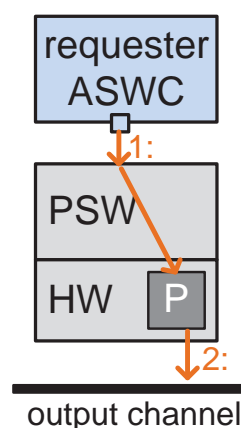


Figure 40: The scenario for accessing an output peripheral covered by this failure model. The following events are shown in the figure: 1: output requested event, 2: output written event.

The output failure model contains seven failure modes. Figure 41 gives an overview of the output failure modes. The failure modes are:

- outFm-1: Digital Output Late
- outFm-2: Digital Output Early
- outFm-3: Digital Output False Positive
- outFm-4: Digital Output False Negative
- outFm-5: Analog Output Late
- outFm-6: Analog Output Early
- outFm-7: Analog Output Value Failure

The writing of digital output is usually performed in a synchronous way. As described in section 4.3.1.2, we usually do not specify timing failure modes for those services. However, if a digital output channel is written by a complex driver, the driver might perform the writing asynchronously. Because of these cases, we have added timing failures to the digital output failure model.

outFm-1: Digital Output Late

A digital output late failure occurs if the delay between the **output requested** event and the **output written** event is too large.

If writing to a digital output is always synchronous, the runtime of a write request should be completely predictable (restrictions apply under some conditions, for example if the memory access times are unpredictable because of unfair memory interconnect arbitration).

In order to specify the tolerable maximum write delay, the digital output late failure is parameterized according to the latency constraint introduced in section 4.3.4 Time.

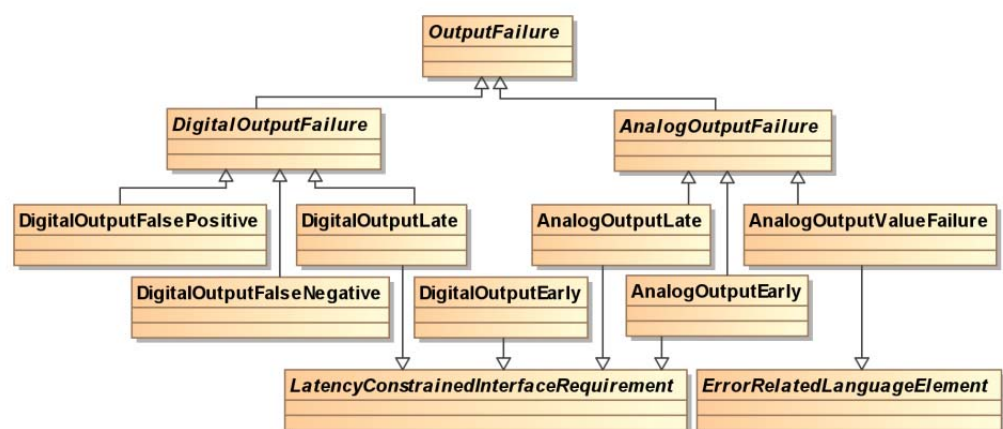


Figure 41: The meta-model of the output failure model

outFm-2: Digital Output Early

A digital output early failure occurs if the delay between the **output requested** event and the **output written** event is too small.

The comment given for outFM-1 regarding the runtime of an output request is valid for this failure mode.

In order to specify the tolerable minimum write delay, the digital output early failure is parameterized according to the latency constraint introduced in section 4.3.4 Time.

outFm-3: Digital Output False Positive

A digital output false positive failure occurs if the Boolean value written to the output channel is **true**, even though the value requested by the requester ASWC was **false**.

outFm-4: Digital Output False Negative

A digital output false positive failure occurs if the Boolean value written to the output channel is **false**, even though the value requested by the requester ASWC was **true**.

outFm-5: Analog Output Late

An analog output late failure occurs if the delay between the **output requested** event and the **output written** event is too large.

In order to specify the tolerable maximum write delay, the analog output late failure is parameterized according to the latency constraint introduced in section 4.3.4 Time.

outFm-6: Analog Output Early

An analog output late failure occurs if the delay between the **output requested** event and the **output written** event is too small.

In order to specify the tolerable minimum write delay, the analog output early failure is parameterized according to the latency constraint introduced in section 4.3.4 Time.

outFm-7: Analog Output Value Failure

An analog output value failure occurs if there is a larger deviation between the actual value on the analog output channel after the **output written event** and the value requested than specified by the acceptable error.

The error can be absolute (e.g., 0,5V) or relative (e.g., 10%). See subsection 4.3.4 Error for more information about modeling errors.

Time Services Failure Model

This subsection describes the failure model of a platform's time services as described in 2.1.4. The time services include a timer service that allows measuring relative time, a global time service that allows accessing a global time base of the system, and a service that allows a runnable entity of the ASWC to wait for a certain period of time.

Measuring a relative time using a timer typically revolves around three different API calls: (1) starting the timer, (2) stopping the timer, and (3) reading the elapsed time. The value returned when reading the elapsed time is the time period since the last start of the timer if the timer has not been stopped since. Otherwise it is the elapsed time between the last start timer / stop timer call sequence. Retrieving a global time, on the other hand, is a straightforward procedure of making a synchronous API call and needs no further explanation. Waiting for a certain period of time also involves a single call. After that call, the task that hosts the runnable transitions into the **waiting** state until the requested time interval is up. After that, the platform has to activate the thread, i.e., set the task's state to **ready**. The scenarios for measuring relative time and for waiting a certain period of time is shown in Figure 42.

The time service failure model contains three failure modes, one for each feature, which are listed below. An overview of the time services failure model is given in Figure 43:

- timeFM-1: Global Time Failure
- timeFM-2: Relative Time Failure
- timeFM-3: Wait Time Failure

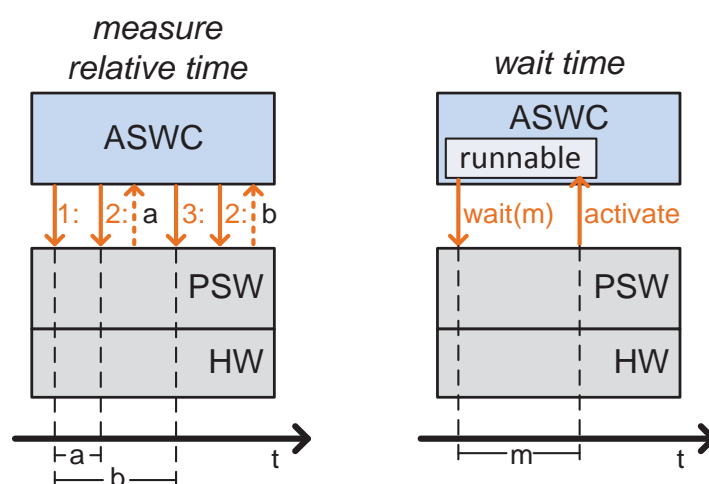


Figure 42:

The scenarios for using time-related services as covered by this failure model. The following events are shown in the figure: 1: start timer, 2: read timer, 3: stop timer.

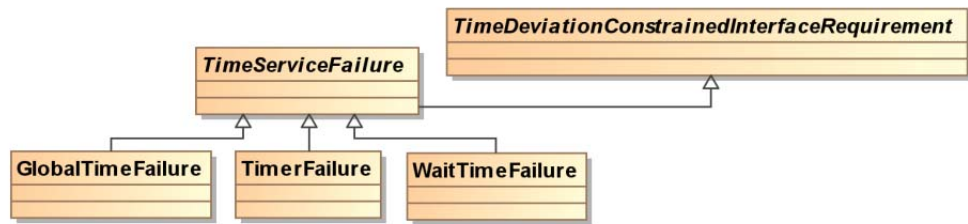


Figure 43: The meta-model of the time services failure model

All failure modes of the time service failure model are parameterized by time deviation constraints (see 4.3.4 Time) to specify the tolerable deviation from the correct time and the time produced by the corresponding service.

timeFM-1: Global Time Failure

A global time failure occurs if the deviation between the correct global time according to the platform's specification and the time delivered as a result by the global time service is bigger than the predefined maximum deviation.

timeFM-2: Relative Time Failure

A relative time failure occurs if the deviation between the correct time between the `read timer` call and the last `start timer` call, or the correct time between the last `start timer`, `stop timer` sequence, and the time returned by a read time call is bigger than the predefined maximum deviation.

Start or stop timer omissions/commissions manifest themselves as relative time failures at the platform API. Consequently, we did not include these failures in the relative time failure model.

timeFM-3: Wait Time Failure

A wait time failure occurs if the time between the `wait time` call and the activation of the runnable deviates from the correct time interval by more than the predefined maximum deviation.

Memory Service Failure Model

This subsection specifies the failure model for a platform's memory services as introduced in subsection 2.1.4. Memory services allow the application to access memory that cannot be directly accessed via the CPUs memory bus, like a flash device connected via an on-board bus, and provide convenience services for memory access, like a file system or

a comparable service as, for example, provided by the AUTOSAR NV-Ram Manager [74].

The memory access service failure model differentiates between read and write scenarios. Both scenarios begin with the application requesting a read or write job. Typically, this job is not processed synchronously but stored in a job queue since the corresponding service or the memory itself might be busy and the process of reading or writing takes several CPU cycles. Therefore, processing the job itself takes a certain period of time as well. When a read job is finished, the data that have been stored in the corresponding memory address are returned to the ASWC. If the write job finishes, there might be a notification to identify the ASWC. This process for using memory services is depicted in Figure 44.

In some platforms, there are also services for managing virtual memory, memory mapping, or caches. These kinds of services are not regarded by our language.

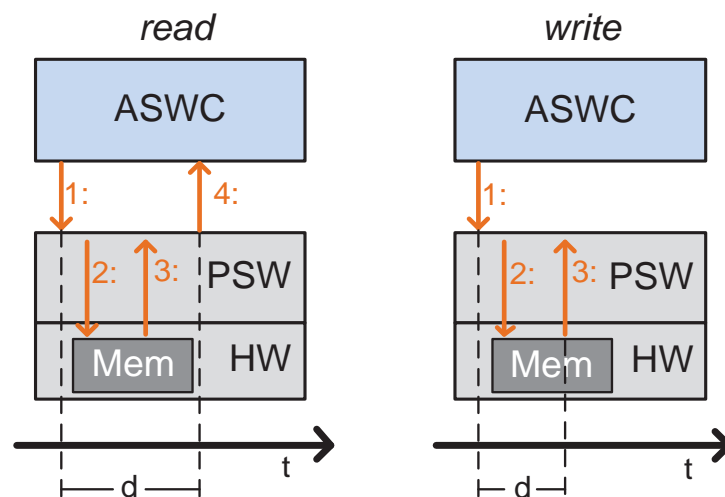


Figure 44:

The scenario for accessing indirectly accessed memory : For reading (left side) and writing (right side). The following events and intervals are shown in the left figure (read): 1: read requested, 2: read started, 3: read finished, 4: result returned, *d*: read delay. The following events are shown in the right figure (write): 1: write requested, 2 write started, 3: write finished, *d*: write delay.

The memory access failure mode consists of six failure modes, listed in the following and depicted in Figure 45:

- memFM-1: Memory Late Read
- memFM-2: Memory Read Access Denial
- memFM-3: Memory Read Data Failure
- memFM-4: Memory Write Delay
- memFM-5: Memory Write Access Denial
- memFM-6: Memory Write Data Failure

memFM-1: Memory Late Read

A memory read delay occurs if the time between the **read request** event and the **result returned** event is larger than a predefined threshold.

In order to specify the tolerable memory read delay, the memory read delay failure is parameterized according to the latency constraint introduced in 4.3.4 Time.

There are no early failures in the memory read failure model, since we are convinced that an earlier read than expected will never be safety critical.

memFM-2: Memory Read Access Denial

A memory read access denial occurs if the service denies the read request to the memory address even though the requester ASWC is allowed to read from that memory address, or if the service accepts the request but does omit to return data.

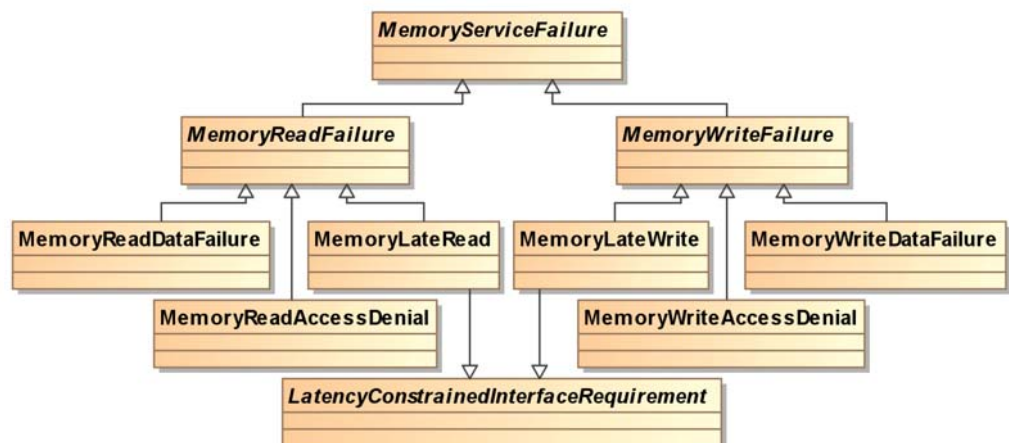


Figure 45: The meta-model of the memory service failure model

memFM-3: Memory Read Data Failure

A memory read data failure occurs if the data that are returned to the ASWC are not identical to the data that were written to the provided memory address by the last write access.

This could mean that either the data have been corrupted since the last write access (e.g., by other applications or by an SEU) or that the read access does not return the data that were stored in the respective memory address when the read was requested.

memFM-4: Memory Write Delay

A memory write delay occurs if the time between the **write request** event and the **write finished** event is bigger than a predefined threshold.

In order to specify the tolerable memory write delay, the memory write delay failure is parameterized according to the latency constraint introduced in 4.3.4 Time.

There are no early failures in the memory write failure model, since we are convinced that an earlier write than expected will never be safety critical.

memFM-5: Memory Write Access Denial

A memory write access denial occurs if the service denies write access to the memory address even though the requester ASWC is allowed to write into that memory address, or if the service accepts the request but omits to write data.

A write to a wrong address would result in a denial of the intended write request and a corruption of the data at the address that have been written.

memFM-6: Memory Write Data Failure

A memory write data failure occurs if the data provided to the service are not identical to the data that are stored at the provided memory address directly after the write process has finished.

Scheduling Failure Model

The scheduling service of a platform is responsible for managing the concurrent access of all ASWC to the platform's CPU or CPUs. Several tasks, and therefore the runnables that run in the context of the task, compete for this shared resource. Scheduling is successful if each time-critical runnable meets its specified deadline.

In accordance with most scheduling models and referring to [75], we define a task's timing model using three variables. The task request time r_i of task T_i is the point in time when the task execution is requested. From this point in time, the task must finish its execution within a certain time interval d_i , the task's deadline. The third variable is the task's (worst case) execution time e_i , which defines the time the task needs to complete after it has been activated. If e_i is correct, a task needs to be activated the latest after the time interval $l_i = d_i - e_i$, which is called the laxity of the task.

From the point of scheduling, it is important to differentiate between event-triggered and periodic time-triggered tasks. An event-triggered task is triggered by a sporadic event, and triggering the task is therefore not directly the platform's job²⁵. A periodically triggered task, on the other hand, is re-requested periodically after a specified time interval p_i . As soon as the first task request r_i is made, every future task request is automatically made by the platform's operating system.

To summarize: The scheduling requirements of a time-triggered task T_i are specified by the triple $T_i = (p_i, d_i, c_i)$, whereas the event-triggered task T_j is specified by the tuple $T_j = (d_i, c_i)$. Figure 46 shows an overview of the events and time intervals relevant for the scheduling process.

As a third function besides scheduling of time- and event-triggered tasks, we consider the execution of interrupts as part of the scheduling function. Unlike tasks, which are requested by software, interrupts are requested by hardware. After the hardware has requested an interrupt, an interrupt controller will eventually make the CPU call the software that handles the interrupt, which is called interrupt service routine (ISR). Other than that, an interrupt can be regarded much like an event-triggered task and indeed, event-triggered tasks are often requested from within an ISR. The time between an interrupt request and the interrupt being served, i.e., the activation of the ISR, is called the interrupt latency. This latency depends on multiple aspects, including the design of the hardware and the operating system, as well as interrupt masking policies and how they are enforced.

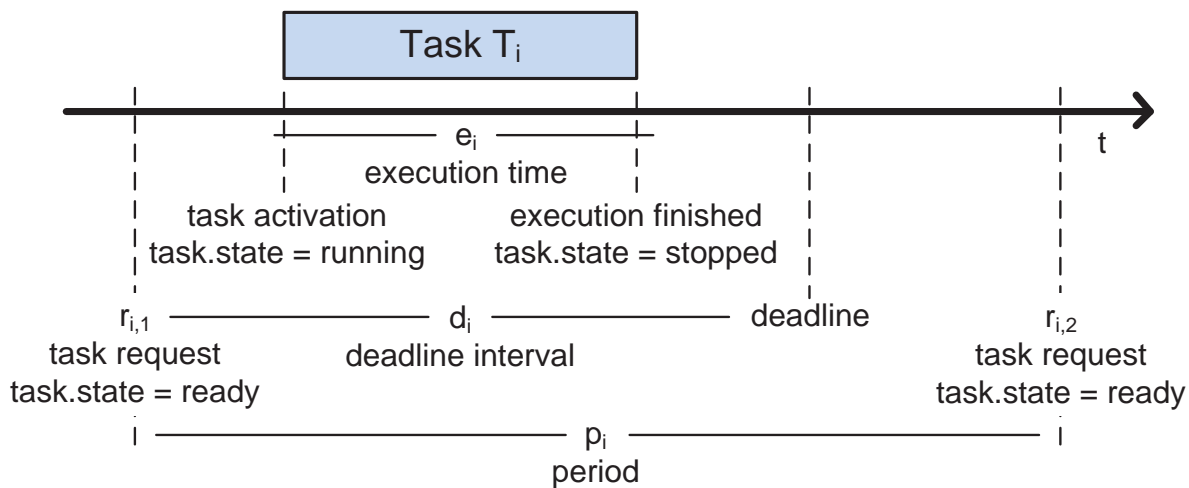


Figure 46: Events and intervals for task scheduling. The period p_i is only relevant for periodically triggered tasks.

²⁵ It is, of course, the platform's job to request the task after the event has occurred, and some sporadic events might also be triggered by the platform.

Regarding the scheduling of time-triggered tasks, event-triggered tasks, and interrupts, the scheduling failure model consists of the following three failure modes, also in Figure 47:

- schedFM-1: Scheduling Jitter Failure
- schedFM-2: Scheduling Deadline Failure
- schedFM-3: Late Interrupt Execution

Figure 47 depicts two classes of runnable-related scheduling failures: **TimeTriggeredRunnableSchedulingFailure** and **GeneralRunnableSchedulingFailure**. Each failure mode inheriting from the first class is only relevant for time-triggered runnables, whereas each failure mode inheriting from the second class is relevant for every kind of runnable (event- and time-triggered).

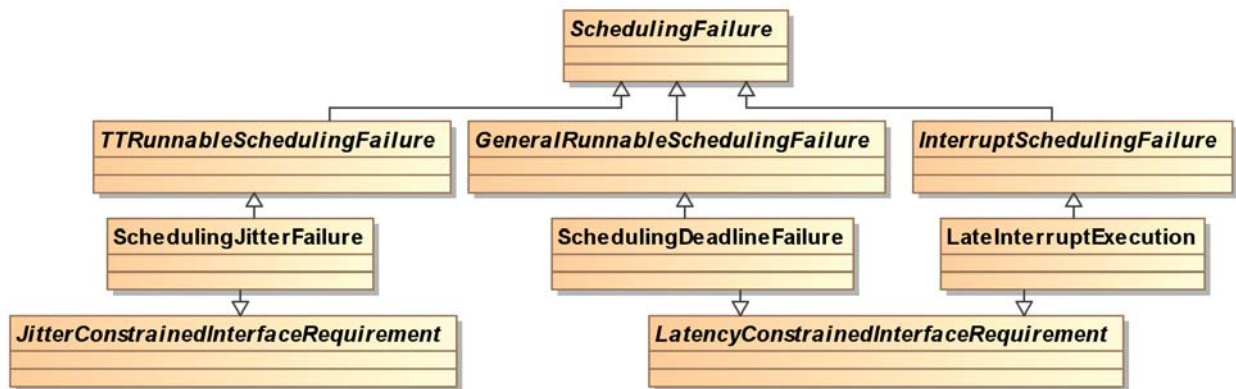


Figure 47: The meta-model of the scheduling failure model

Prior to the introduction of the failure models, we want to make some comments regarding the design of the scheduling failure model. First of all, the application language is meant to contain demands regarding the behavior of the platform. Without additional measures, the platform alone cannot guarantee compliance with scheduling demands, since this depends on the execution time of the application. Therefore, the presented scheduling failure modes do not cover the failure of another runnable exceeding its WCET, meaning that every guarantee or demand given using these failure modes implicitly assumes that the WCETs are correct. In case the application requires protection from other ASWCs exceeding their execution time, a protection demand has to be specified. Protection demands are introduced in section 4.5.3, protection guarantees in section 4.6.3.

Having said this, we must also note that the WCET of a runnable is not completely in control of the runnable. The WCET depends on many things, such as the context switching latency of the OS, the WCET of the platform services used, and the performance of the computational resources, like CPU or memory. In the current version of the VerSal

language, demands regarding the trustworthiness of WCET computation are not covered by our language.

schedFM-1: Scheduling Jitter Failure

A scheduling jitter failure occurs if the time between two successive **execution finished** events of the same runnable deviates by more than a predefined maximum value.

In order to specify the tolerable jitter, the scheduling jitter failure is parameterized according to the jitter constraint introduced in section 4.3.4 Time.

schedFM-2: Scheduling Deadline Failure

A scheduling deadline failure occurs if the runnable finishes its execution after its deadline has expired (the time between **runnable requested** and **runnable finished** is bigger than d_i).

In order to specify the deadline, the scheduling deadline failure is parameterized according to the latency constraint introduced in section 4.3.4 Time.

schedFM-3: Late Interrupt Execution

A late interrupt execution failure occurs if the time between **interrupt requested** and **execution of ISR** is larger than a predefined maximum delay.

In order to specify the maximum interrupt latency, the late interrupt execution failure is parameterized according to the latency constraint introduced in section 4.3.4 Time.

Basic Execution Failure Model

So far the failure model has been covering special-purpose services of a platform. The basic execution failure model covers the failure modes of platform hardware components that have a crosscutting effect on almost any service a platform provides. Therefore, we suggest that every safety-critical application specifies a detection or avoidance demand for each of the failure modes listed below (this can also be done automatically by our tool).

This failure model contains four failure modes, which are also depicted in Figure 48:

- basExFM-1: CPU Failure

- basExFM-2: Main Memory Failure
- basExFM-3: CPU Clock Failure
- basExFM-4: Power Supply Failure

basExFM-1: CPU Failure

A CPU failure occurs if the behavior of the CPU deviates from its specification.

basExFM-2: Main Memory Failure

A main memory failure occurs if the behavior of the main memory (directly accessed memory like cache, RAM, flash) deviates from its specification. Such a failure can also be caused by hardware components like caches or interconnects used to connect the CPU to the main memory.

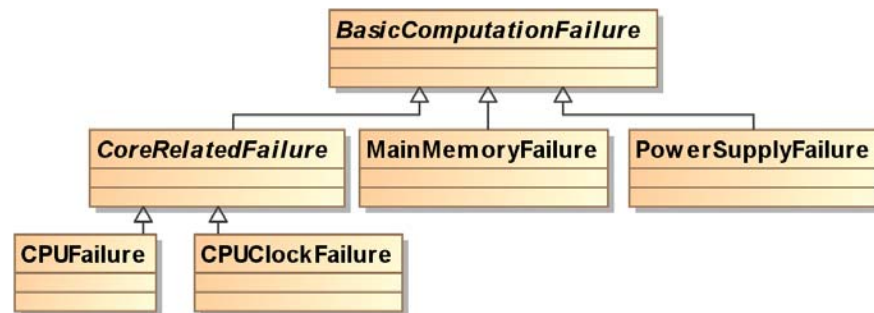


Figure 48: The meta-model of the basic computation failure model

basExFM-3: CPU Clock Failure

A CPU clock failure occurs if the behavior of the CPU clock deviates from its specification (i.e., runs faster, runs slower than specified or stops).

basExFM-4: Power Supply Failure

A power supply failure occurs if the behavior of the power supply deviates from its specification (i.e., voltage drops, rises, fluctuates or the supply fails completely).

4.4.3 Application Failures

The application failure package contains the failure modes of an application that can be detected by a platform. Unlike the platform service failure modes, the application failure modes were not identified by a failure analysis of an application. First, this would not be possible since applications have no standardized behavior like standardized

execution platforms. Second, analyzing an application regarding its failure modes would yield every failure mode and not only those that can be potentially detected by a platform. Therefore, the platform-detectable application failures were derived from standard detection mechanisms available for typical platforms.

As a result of the analysis, this section introduces five different application failure modes that can be detected by appropriate monitoring facilities. These are:

- appFM-1: Arrival Rate Failure
- appFM-2: Inter-Arrival Time Failure
- appFM-3: Execution Time Deviation
- appFM-4: Logical Sequence Failure
- appFM-5: Runtime Failure

The first three and the last failure modes describe deviations from the nominal timing-related behavior of an application. The fourth failure mode describes a deviation from the application's logical behavior. Since a multi-purpose execution platform does not per se know the nominal behavior of its guest applications, the platform's monitoring facilities must be adaptable, so that an integrator can enable them to detect the application's failures by configuring them appropriately.

A look at the list of failure modes reveals that there is no demand regarding the detection of deadline or time-window misses. This is because demands regarding the detection of these failures are specified using the scheduling failure.

Every application failure mode specified in this package is observed on the runnable level. Therefore, every application failure mode has an architecture reference to a supervised element, which, on the application level, points towards a runnable. Figure 49 shows the corresponding meta-model including all platform detectable failure modes and their architecture references.

A runnable is the smallest executable entity of an application, and consequently, higher-level failures (for example on the ASWC level) can usually be inferred by the application using runnable-level failures. If a specific type of platform provides detection mechanisms on other abstraction levels, this part of the VerSal language must be extended or adapted.

appFM-1: Arrival Rate Failure

An arrival rate failure occurs when the number of times a runnable is executed during a certain time interval exceeds or falls below a specified threshold.

To specify the minimum and maximum arrival rates, the arrival rate failure contains two integer parameters (**min** and **max**). To specify the supervision interval, the arrival rate failure contains a time parameter (**interval**).

Arrival rate monitoring can be used to check for the aliveness of a runnable. It is possible to check whether activation of the runnable is too frequent or too scarce. The arrival rate is the reciprocal of the inter-arrival time, as it measures the count of runnable activations per time unit. The arrival rate is typically a more coarse-grained measure for the aliveness of a runnable than the inter-arrival time, but its implementation is also less resource consuming since the arrival rate has to be checked only once per time interval.

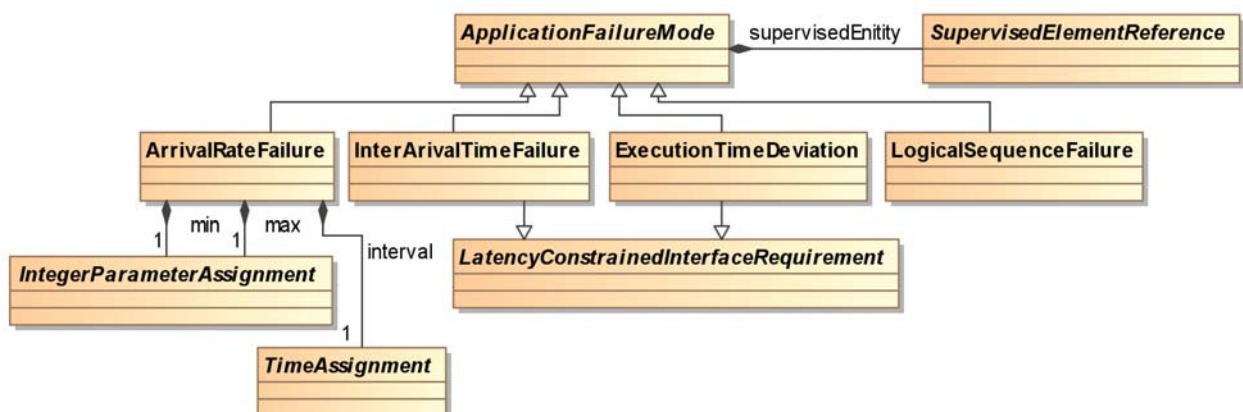


Figure 49: An excerpt of the meta-model for application monitoring demands

appFM-2: Inter-Arrival Time Failure

An inter-arrival time failure occurs when the time between two executions of a runnable exceeds or falls below a specified threshold.

In order to specify the acceptable inter-arrival time, the inter-arrival time failure is parameterized according to the latency constraint introduced in section 4.3.4 Time.

Inter-arrival time monitoring can be used to check the aliveness of a runnable, but also to check its periodicity. It is possible to check whether the inter-arrival time is too long or too short. The arrival time is the reciprocal of the arrival rate. Arrival time monitoring is typically a more precise measure for the aliveness of a runnable, but it is also more

resource consuming since it has to be checked every time a runnable is activated.

appFM-3: Execution Time Deviation

An execution time deviation occurs when the time between the activation of a runnable and the termination of the runnable exceeds or falls below a specific threshold.

In order to specify the acceptable execution time, the execution time deviation failure is parameterized according to the latency constraint introduced in section 4.3.4 Time.

Execution time monitoring evaluates the time between the activation of the runnable and the termination of the runnable. It is better suited for detecting the root cause of a deadline miss than deadline monitoring. Execution time monitoring is able to exactly identify the runnable that exceeded its execution time. Conversely, when detecting a deadline miss, it is impossible to say whether the runnable that missed its deadline caused the scheduling failure or any runnable that was scheduled before.

appFM-4: Logical Sequence Failure

A logical sequence failure occurs when the sequence of points in the control flow of the runnable deviates from the possible sequences of a nominally working runnable.

The valid logical sequences of a runnable are typically modeled as a graph of checkpoints. If there is no edge between the previous and the actual checkpoints, a logical sequence error is detected.

appFM-5: Runtime Failure

A runtime failure is an exception that is raised by the general computation hardware or by the platform software during the execution of the application. Examples of hardware-raised runtime failures are divisions by zero, execution of kernel-level instructions in user-mode, or access to a protected memory region. Typical runtime failures raised by the platform software are invalid access to protected services, use of uninitialized services, or invalid service calls (e.g., termination of a task while blocking a shared resource).

The runtime failures available on a specific platform depend on the microarchitecture and the platform software (OS and middleware components).

4.4.4 Platform Failure Reactions

This section introduces the standardized platform failure reactions provided by the VerSal language. A platform failure reaction is a reaction that the platform is able to perform whenever a failure occurs. In the context of safety-critical systems, it is common to use the platform to perform failure control since the application itself might be unreliable or unable to perform a reaction after it has encountered a failure. Furthermore, some failure reactions like shutting cannot be performed by the application due to a lack of sufficient permissions.

Comparable to the application failures introduced in section 4.4.3, the platform failure reactions presented in this section were also identified by analyzing common standardized platforms, like AUTOSAR or IMA 653 compliant platforms. During our analysis we identified six types of platform failure reactions. These are:

- Restart
- Shutdown
- Write Default Signal
- Send Default Message
- Indication
- Handler Execution

Restart and shut down are generalized reaction types. In order to completely specify a shutdown or a restart, the application developer needs to specify the object that is restarted or shut down. The options are: (A) the task hosting the runnable that is referenced by the demand, (B) the partition hosting the ASWC that is referenced by the demand, or (C) the complete platform. The corresponding excerpt of the platform failure reaction meta-model is shown in Figure 50.

Restart

Restart is an often used measure to react to a detected failure. A restart basically deletes the current state of the system and puts the system into a pre-defined initial state. If the system detects a failure, it is sometimes unclear if the failure corrupted the state of the system and/or how to repair the corrupted state. In such a case, a restart resets the system state. If the failure was transient, the system might behave normally after the restart.

We differentiate between three levels of restart. On the first level, we allow demands regarding the restart of tasks. Restarting a task resets all runnables that are executed in the context of the task. On the second level, there are demands regarding the restart of a partition. A restart of a partition will result in a restart of all tasks running in the context of the

partition as well as in a reset of all resources held by the partition. On the last level, we allow demand for a restart of the platform. Such a restart typically involves a restart of the MCU as well as a restart of all stateful devices.

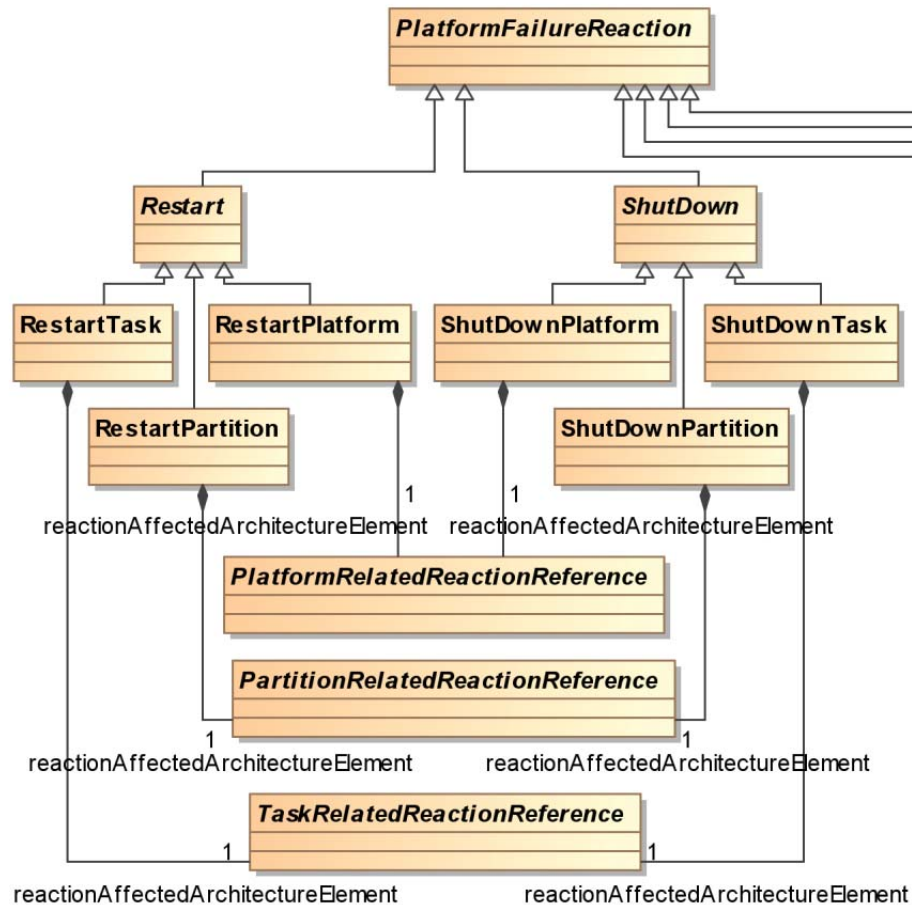


Figure 50: An excerpt of the platform failure reaction meta-model . The excerpt shows the part of the model that specifies restart and shutdown reactions.

Shutdown

Like restarting, shutting the system down is a measure to roll the system forward to a known state. However, in case of a shutdown the target state is not the initial but the final state of the system. A shutdown is typically performed if a failure is detected but there is reasonable suspicion that the cause of the failure is permanent (e.g., if the system has been restarted several times). Shutting the system down is a standard means to put fail-safe systems into their safe state.

Again, we differentiate between three levels of restart. On the task level, all runnables executed in the context of the task are shut down. On the partition level, a shutdown affects all tasks running in the context as well as all resources held by the partition. On the platform level, the MCU and all devices are shut down.

Write Default Signal

Writing a default signal on an output channel is the standard means for bringing an actuator to its safe state. Often, the default signal equals the state that the output channel has if it is de-energized. In this way, the actuator is automatically put into its safe state if the platform is shut down. However, putting the actuator into a safe state is not only performed in the course of an overall shutdown but can also be demanded if, for example, the ASWC driving the actuator fails. In such a case, we allow demanding to write a default signal on a physical output channel using the write default signal demand.

We differentiate between digital default signals and analog default signals. A digital default signal can have the default value set to logical zero or logical one, whereas an analog default signal can have a floating point value combined with a unit (represented by a semantics-less string).

Send Default Message

Sending a default signal via a communication channel is typically done to inform other communication participants about a failure. In case the application is unable to send the corresponding default message, the platform can be used to perform this task, which is specified using the send default message reaction.

The following EBNF production rules are used to specify demands regarding the sending of default messages. The default message itself is only represented by a semantics-less string. We show the example production rule for a simple reaction demand to send a default message. The corresponding complex reaction demand is designed analogously.

Indication

Indicating a failure is typically done when the ASWC is able to control the failure by itself. In such a case, the platform first detects the failure and then indicates the failure to the ASWC. The action to control the failure is finally performed by the ASWC.

Handler Execution

Comparable to an indication reaction, the handler execution reaction leaves the control of the failure to the application. But instead of indicating the failure, the platform executes a pre-defined error handler, a program written by the application developer. Compared to an indication reaction, the handler execution allows for a much quicker failure reaction time since the program controlling the failure does not need to be scheduled first.

4.5 Application Language

In this section, we will introduce the application language. The application language is used by the application developer to specify the safety-critical demands regarding the behavior of the platform and is the counterpart to the platform language, which will be introduced in section 4.6. The specification of the application language uses the common types, attributes, and relations that are part of the common language introduced in sections 4.3 and 4.4; therefore, we recommend reading the corresponding sections before studying the application language.

The application developer uses the application language to specify the vertical safety interface of the application. This safety interface contains all the assumptions regarding the behavior of the application's host platform(s) that have to be met to validate the application's safety case. If one of these assumptions is not met by a platform, the application safety case loses its soundness and deploying the application to such a platform may result in an unsafe system. To stress their critical nature, we call those assumptions demands. On the other hand, if all demands are met by the host platform(s) and the application safety interface is correct and complete, the application will execute safely. Correctness and completeness of the application safety interface have to be assessed and, where appropriate, certified before it can be used for credible VerSal mediation. Such an assessment shall conclude that the application is fit for safety-critical execution on an execution platform, provided the execution platform fulfills all demands specified in the application's vertical safety interface.

As mentioned briefly in section 4.2, the structure of the application and platform language is based on the observation that there are four major classes of safety-related dependencies between an application and a platform. Analogously, the application language consists of four basic types of identically named demands. These are: platform service demands, health monitoring demands, resource protection demands, and service diversity demands.

A *platform service demand* enables the application developer to specify demands regarding the detection or avoidance of platform failures that would otherwise affect the safe behavior of the application. In contrast to seeing the platform's role as a cause of failure, it is common practice to use the platform as a means for detecting application failures and for executing failure containment reactions. To specify the corresponding demands, the application developer uses the so-called *health monitoring demand*. In addition to these failure-centric demands, we also have to cover the challenges posed by mixed-critical applications that share common platform resources and services. To protect highly critical

applications from application with lower criticality, the platform must provide certain protection mechanisms. The application developer demands the availability of these mechanisms using *resource protection demands*. Finally, the last demand class allows specifying *service diversity demands*, which enable the application developer to demand that different services are developed diversely/dissimilarly. This, on the other hand, enables the application developer to use so-called integrity level decompositions in the application-level safety case and to specify the resulting demands in the vertical safety interface.

Figure 51 shows the top-level structure of the application language meta-model including the different demand classes. You might realize that the vertical application (safety) interface does not contain but reference the application demands. This is because the application demands are contained in appropriate architecture elements of the application (see section 4.3.3 for more information on demand containment). The mapping between demands and container elements will be introduced in the next subsection.

The following subsections are ordered as follows: Platform service demands are introduced in subsection 4.5.1, health monitoring demands in subsection 4.5.2, resource protection demands in subsection 4.5.3, and service diversity demands in subsection 4.5.4.

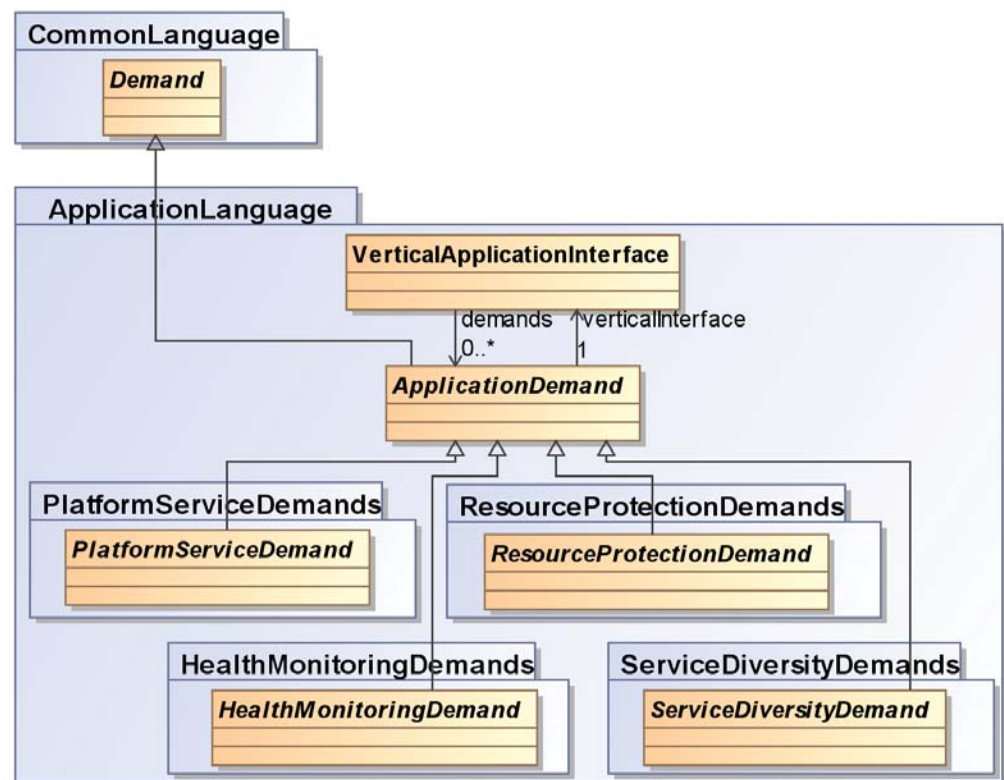


Figure 51: The top-level meta-model of the application language

4.5.1 Platform Service Demands

A platform service failure demand enables the application developer to specify demands regarding the avoidance or detection of platform service failures. It is the first in a series of four top-level demand classes that constitute the application language.

Providing infrastructural services to an application is the primary task of every execution platform. The application requires these services in order to realize its functions. Consequently, the correct provision of a function directly depends on the correct provision of the platform services required to realize the function. Therefore, failure of a platform service may lead to failure of all functions relying on the service. If one of these functions is safety-critical, the consequence is that the platform service becomes safety-critical, too. Platform service demands allow the application developer to view a platform as a potential source of failure and to specify demands regarding these.

Whenever the application developer uses a platform service, he or she has to perform the following steps. First, the effects of the service's possible failure modes on the behavior of the application have to be analyzed. To do so, the application developer uses the standardized failure modes presented in section 4.4.2, for example as a starting point of an FMEA analysis or as a possible basic failure event in an FTA. In case a failure mode might lead to a safety-critical failure of the application, the application developer has four options: (1) redesign the application so that the failure is no longer safety critical; (2) introduce mechanisms into the application that detect and control the failure mode in an appropriate way; (3) specify a platform service demand that requests the platform to detect the failure and either indicate it and leave the control to the application, or perform a failure reaction (this is done using health monitoring demands, see section 4.5.2); (4) specify an avoidance demand. To fulfill an avoidance demand, the platform has to control the failure so that there are no visible negative effects for the application. Figure 52 gives an overview of an application developer's design decision when encountering a safety-critical platform service failure.

Figure 52 shows that when the application developer decides to specify a platform service demand, the main decision is whether to specify a detection or an avoidance demand. Therefore, we continue this section by providing further details about the syntax and semantics of detection and avoidance demands.

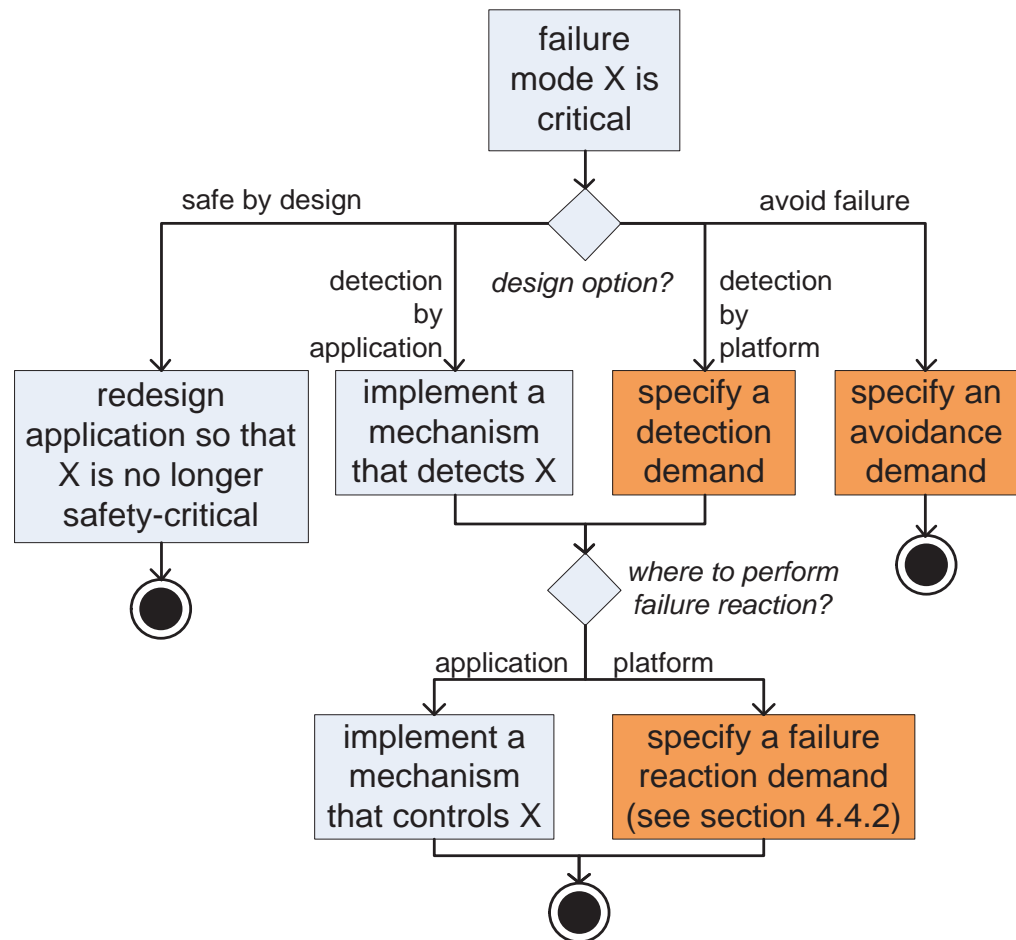


Figure 52: Flow chart showing the service failure demand design options of an application developer when encountering a platform service failure. Orange rectangles indicate that the specification of an application demand is necessary.

A detection demand is fulfilled if the platform has the capability of detecting the corresponding failure. Since detecting a failure is never sufficient for controlling a failure, a detection demand must always be accompanied by at least one appropriate reaction demand. If no reaction demand is specified by the application developer, a default reaction demand is generated, which demands that the detected failure must be indicated to the application. This enables the application to control the detected failure. If the application is unable to control the failure itself, the application may use the failure reactions provided by the platform, which are specified using health monitoring demands. If that does not yield a safe application either, the only remaining option is to specify an avoidance demand.

Furthermore, when specifying a detection demand, the application developer has to specify the demanded failure detection time. The *failure detection time* is the time between the occurrence of the failure and the starting point of the failure reaction. If the application developer specifies a failure detection time of "0 ms", the corresponding failure must be detected before it is able to affect the behavior of the

application. This is, for example, possible for any kind of data corruption (e.g., signal corruption). In this case, the corrupted signal must be detected before the application is able to use the data.

An avoidance demand is fulfilled if the chance for the corresponding platform service failure to occur is acceptably low. Since the VerSal language focuses on systematic failures, the criterion for acceptability is specified using integrity levels. Compared to detection demands, avoidance demands are usually harder to fulfill. Consequently, when specifying an avoidance demand, the chance for successful mediation decreases. Furthermore, when an application developer decides to specify an avoidance demand, there are two important issues regarding its semantics: First, an avoidance demand is not absolute. To fulfill an avoidance demand, there must be mechanisms in place that can prevent the occurrence of the failure. Yet there might still be a certain probability or some special scenarios where the failure mode cannot be avoided. Deciding upon the sufficiency of an avoidance mechanism is done during the interface mediation step and is based on the integrity level of the avoidance demand. Second, an avoidance demand is no correctness demand. The platform can fulfill an avoidance demand by transforming the corresponding failure into another failure mode when it occurs. A demand to avoid a signal corruption can, for example, be implemented by detecting the signal corruption, discarding the signal, and thereby transforming the corruption into an omission.

We will now continue with a detailed description of the modeling of platform service demands, including additional parameters, the integration of the demands into the architecture model of the application, and the usage of the common platform service failure model.

At the root of the platform service demand meta-model is the differentiation between demands for detection (**PlatformServiceFailureDetectionDemand**) and demands for avoidance (**PlatformServiceFailureAvoidanceDemand**). Based on this first differentiation, the meta-model contains different classes for different types of failure demands (e.g., **MutexFailureDetectionDemand**). Each of these failure-mode-specific demands is related to a specific element of the application model and to the corresponding failure modes of the element. Figure 53 shows the related part of the platform service demand meta-model.

Each failure-specific demand is contained²⁶ by a different type of application element (e.g., a mutex) and may contain one of the possible failure modes of the element (e.g., mutex failures). Using this modeling

²⁶ Please note that every demand containment is realized by the containment pattern introduced in section 4.3.4 and depicted in Figure 23.

pattern, an abstract platform service failure mode (e.g., a mutex access commission) is related to a specific architecture element (mutex_1). This turns the abstract failure mode into a failure mode of this specific element (a mutex access commission of mutex_1). This pattern is necessary since an application often utilizes the same service class in several ways, e.g., it uses a communication link to send different signals. Different utilizations of the same service may have different safety requirements (one signal carries critical information, the other does not). Using this modeling pattern, the application developer can specify different demands for different utilizations of the same service. Of course, when instantiating and parameterizing a platform service demand, the application developer must also parameterize the related failure mode (see section 4.4.2 for the failure modes of each parameter) and the integrity level of the demand (a parameter inherited from the abstract interface requirement class, see section 4.3.4 Integrity Level).

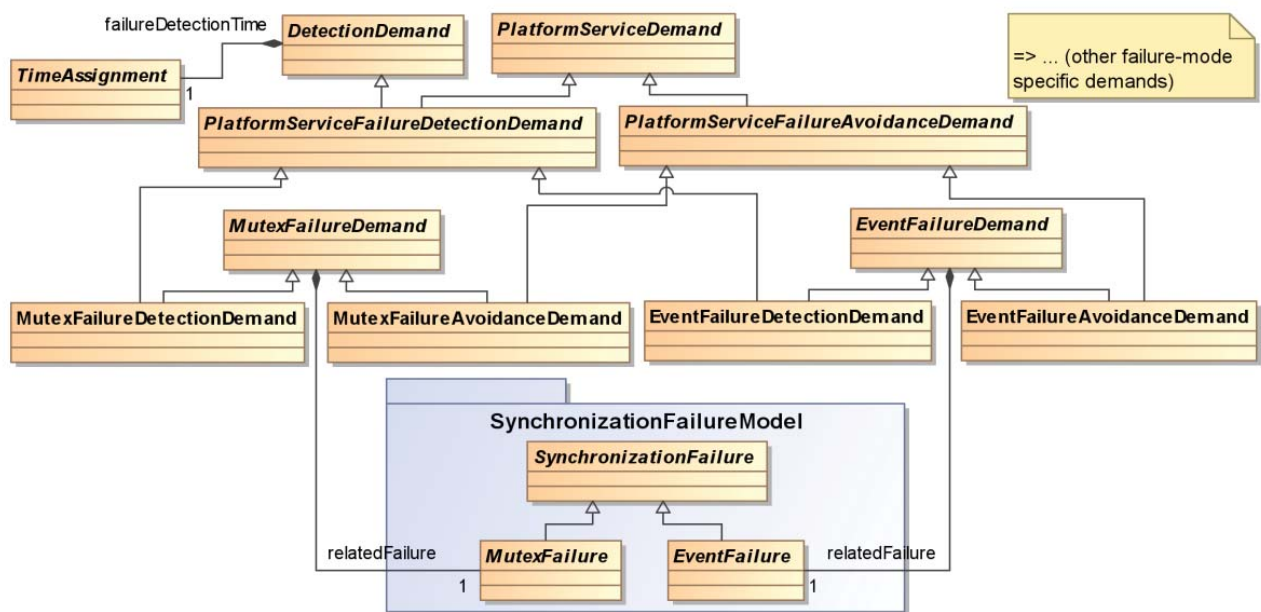


Figure 53: The principal structure of the platform service demand meta-model

Table 7 shows all failure-specific platform service demands including their related application elements and failure modes.

Table 7: A list of all platform service demands including their related application elements and failure modes.

<i>Demand</i>	<i>Related application element</i>	<i>Related failure mode</i>	<i>Failure model</i>
MutexFailureDemand	MutexService-Need	MutexFailure	Synchron.-FailureModel
EventFailureDemand	EventService-Need	EventFailure	“

Communication-FailureDemand	Communication-Port	Communication-Failure	Communication-FailureModel
DigitalInput-FailureDemand	DigitalSensorIn-Port	DigitalInput-Failure	InputFailure-Model
AnalogInput-FailureDemand	AnalogSensorIn-Port	AnalogInput-Failure	“
DigitalOutput-FailureDemand	DigitalActuator-OutPort	DigitalOutput-Failure	OutputFailure-Model
AnalogOutput-FailureDemand	AnalogActuator-OutPort	AnalogOutput-Failure	“
GlobalTimeFailure-Demand	GlobalTime-ServiceNeed	GlobalTime-Failure	TimeService-FailureModel
RelativeTime-FailureDemand	TimerService-Need	RelativeTime-Failure	“
WaitTimeFailure-Demand	WaitService-Need	WaitTime-Failure	“
MemoryService-FailureDemand	MemoryService-Need	MemoryService-Failure	MemoryService-FailureModel
TTRunnableScheduling-FailureDemand	TimeTriggered-Runnable	TTRunnable-Scheduling-Failure	Scheduling-FailureModel
RunnableScheduling-FailureDemand	Runnable	General-Runnable-Scheduling-Failure	“
InterruptScheduling-FailureDemand	ISR	Interrupt-Scheduling-Failure	“
CoreRelatedFailure-Demand	Runnable	CoreRelated-Failure	BasicExecution-FailureModel
MainMemoryFailure-Demand	MemorySection	MainMemory-Failure	“
PowerSupplyFailure-Demand	ASWC	PowerSupply-Failure	“

In the following, we will introduce some example platform service demands based on the running example introduced in section 4.1.

Example D1 presents a failure avoidance demand for a sampling latency failure of the analog input signal `v_raw_A`.

Example D1: A sampling latency of the input signal `v_raw_A` larger than 0.2ms must be avoided (ASIL B).

Example D2 presents a failure detection demand for a value failure of the output signal **a_set_fin**.

Example D2: A value failure of the output signal a_set_fin larger than 0.05V must be detected within 0.05ms (ASIL C).

Example D3 presents a corruption failure of the communication signal **v_ref** without any specification of a failure detection time. The semantics in this case is that the message corruption must be detected before the message reaches the corresponding software component (in this case **v_controller**).

Example D3: A corruption of the signal v_ref must be detected (ASIL C).

4.5.2 Health Monitoring Demands

A health monitoring demand enables the application developer to specify demands regarding the detection of application failures and the execution of failure reaction. This is the second in a series of four top-level demand classes that constitute the application language.

As described in section 4.5.1, the execution platform can be regarded as a source of failure. But it is also common for the platform to take the opposing role and provide safety mechanisms to the application. To some extent, the platform can be regarded as an independent component that is able to keep track of most of the application's actions. This makes the platform a suitable candidate for monitoring application failures. In addition to its partial independence of the application, the platform has many rights for performing actions (like a shutdown) that the application does not have, which allows the platform to provide more powerful failure reactions.

To cover both aspects, the health monitoring demand model is divided into two parts. The first part contains application monitoring demands, allowing the developer to specify demands regarding the detection of application failures. The second part contains failure reaction demands, allowing the application developer to specify demands regarding the execution of failure reactions. Figure 54 gives an overview of the health monitoring model.

In the following two subsections, we will first introduce application monitoring demands and then failure reaction demands.

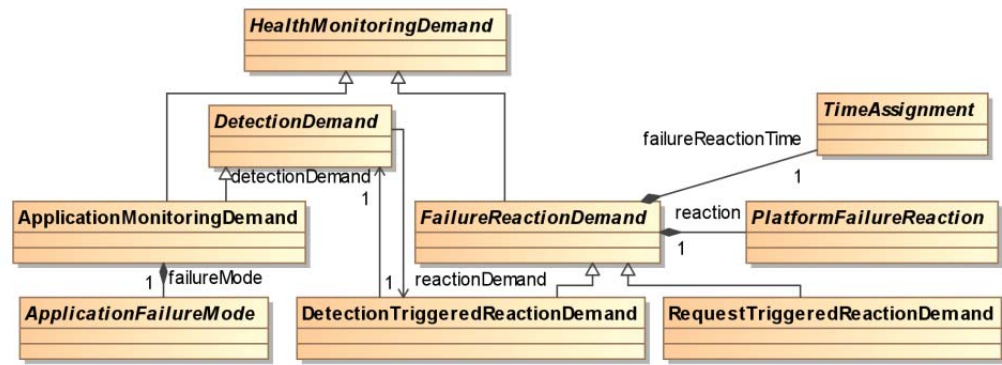


Figure 54: The health monitoring meta-model

Application Monitoring Demands

The application developer specifies an application monitoring demand to strengthen the fault tolerance of the application under development. The application monitoring demand requires the platform to detect a deviation from the application's nominal behavior, i.e., an application failure. The platform has the capability of monitoring the behavior of the application since the platform is involved in the realization of most of the application's functionality. Of course, a plain execution platform cannot generally tell an application's nominal behavior from a failure; hence, the platform provides general-purpose monitoring mechanisms that have to be configured. This configuration is performed by the integrator, which then enables the platform to detect the failures of guest applications.

There are many reasons for an application developer to specify an application monitoring demand. The first is to protect the application from design faults. Some standards demand the application to be monitored if its criticality exceeds a certain level. But application monitoring demands are also suitable to detect failures where the root cause is not found in the application itself. It is also possible that a failure external to the application may cause a failure that is perceived as a deviation of the application's behavior (the omission of another application to trigger an event can easily lead to a timing failure of the supervised application). Therefore, the application developer also specifies application monitoring demands if the application uses untrusted components and the failure of the untrusted component may manifest itself as a detectable application failure.

If the application developer decides to specify an application monitoring demand, an application monitoring demand is instantiated (see Figure 54). Unlike platform service demands, an application monitoring demand is not directly contained in the element that shows the failure mode, i.e., the supervised runnable, but by an application monitoring need (see annex A.2). The model is designed like this because an application monitoring need is always deployed to an application monitoring service

(see annex A.3), allowing the mediation algorithm to directly check the available monitoring capabilities for their sufficiency.

When the demand is instantiated, the developer chooses an application failure mode from the application failure model specified in section 4.4.3. As described in section 4.4.3, application failures reference an executable unit of the corresponding application to identify the supervised entity. Where applicable, the application developer has to parameterize the failure mode chosen. The parameters of the application failure modes are also specified in section 4.4.3. Since avoiding application failures does not lie within the power of the application, every application monitoring demand is automatically a detection demand, which requires the application developer to specify a failure detection time (see section 4.5.1 for more information regarding failure detection time).

In the following, we will introduce some example application monitoring demands based on the running example introduced in section 4.1.

Assume that the `v_controller` software component contains one executable/job called `v_controller_main` that has an execution time of `0.015ms`. Example D4 specifies an application monitoring demand that demands the detection of deviations from this nominal execution time with a small safety margin of `0.001ms`.

Example D4: The platform must detect an execution time of the executable `v_controller_main` of more than `0.016ms` (ASIL C).

Example D5 specifies another application monitoring demand that requires the supervision of the logical execution sequence of `v_controller_main`. The prerequisite for configuring the corresponding monitoring mechanism is specified in the corresponding guarantee (see Example G4).

Example D5: The platform must detect a logical sequence failure of the executable `v_controller_main` (ASIL C).

Failure Reaction Demands

The application developer specifies a failure reaction demand to request a failure controlling reaction from the platform. In an integrated architecture, the application is only allowed to use the platform API to interact with the platform software and hardware. This design restriction protects the platform and other applications from erroneous applications, but also limits an application's freedom to perform certain failure recovery reactions. Therefore, an application uses the platform to explicitly trigger these restricted recovery reactions. In addition to the

extended permissions of the platform, the platform can be seen as an independent element to the application. When an application-related failure occurs, this may render the affected application unable to perform a reaction, whereas the platform might still be able to react. Consequently, the platform can be used to perform recovery reactions when the reliability of the application is in question.

As stated above, there are two reasons for an application developer to specify a failure reaction demand: (1) if the application has insufficient rights to perform the required reaction, and (2) if, as a consequence of a failure, the application is unable or unreliable to recover from the failure itself. Depending on whether the application is capable of performing or triggering recovery reactions in case of failure, the VerSal language offers two types of failure reaction demands: request-triggered reaction demands and detection-triggered reaction demands. The application developer uses a request-triggered reaction demand if the application is capable of triggering the reaction but is unable to autonomously perform the recovery reaction. A request-triggered reaction is performed by the platform but triggered by the application by calling the corresponding API function. By using this kind of reaction demand, the application is in full control of starting the recovery reaction.

The application developer uses a detection-triggered reaction demand if the application is not reliable enough to trigger the reaction itself. In case of a detection-triggered reaction demand, the platform automatically performs the reaction as soon as the related failure occurs. If it uses detection-triggered reaction demands, the application has no responsibility in the process of starting or executing the failure recovery. However, this kind of demand can only be used if the platform is aware of the failure that triggers the reaction, which restricts these kinds of demands to platform service failures and platform-detectable application failures. Failures that are internal to an application cannot automatically trigger platform reactions. But since internal application failures are detected by the application, it is safe to assume that the application is also able to reliably trigger the appropriate platform failure reaction.

When the application developer specifies a failure reaction demand, no matter which type, he or she specifies the failure recovery reaction the platform has to perform. This is done using the common recovery reactions specified by the common platform failure reactions (see section 4.4.4). When a failure reaction contains parameters, the developer must also configure the corresponding reaction appropriately.

Furthermore, every reaction demand requires the specification of a failure reaction time. A failure reaction time specifies the maximum time interval that may elapse between triggering the recovery reaction and finishing the execution of the reaction. The application developer uses the failure reaction time parameter together with the failure detection

time parameter to ensure that the time between failure occurrence and failure recovery is smaller than the failure tolerance time given by the physical process controlled by the application. In case the failure tolerance time is particularly small, the application developer might consider using detection-triggered reaction demands rather than request-triggered reaction demands, as this type of demand usually allows for quicker reaction time.

In case the application developer specifies a detection-triggered reaction demand, he or she must specify which failure detection triggers the reaction. This is done by referencing a corresponding detection demand, i.e., a platform service detection demand or an application monitoring demand that is already specified. This identifies the trigger condition of the automatic recovery reaction and assures that the application developer specifies a failure detection triggered reaction demand only if the corresponding detection demand is specified as well. If the application developer wants to specify that several failures trigger the same reaction (e.g., a standard shutdown), several detection triggered reaction demands must be specified.

In the following, we will introduce some example failure reaction demands based on the running example introduced in section 4.1.

Examples D6 and D7 show two request-triggered reaction demands owned by the software component **monitoring**. This software component is capable of detecting deviations in the redundantly measured vehicle velocity and demands the restart of both sensor software components in case such a deviation is detected.

Example D6: Upon request, the platform must restart the task hosting executable v_sensorSWC_A_main (ASIL C).

Example D7: Upon request, the platform must restart the task hosting executable v_sensorSWC_B_main (ASIL C).

Example D8 specifies a detection-triggered reaction demand owned by the software component **throttleSWC**. This software component demands that upon detection of the previously specified output value failure of **a_set_fin** (see Example D2), the platform automatically sets the output signal to its safe default value. Please note that the output value failure is unambiguously defined. The corresponding detection demand that triggers the reaction is referenced in the meta-model (see **detectionDemand** relation in Figure 54).

Example D8: Upon detection of the output value failure of signal a_set_fin, the platform must set the signal to the default signal 0.0V within 0.05ms (ASIL C).

4.5.3 Resource Protection Demands

A resource protection demand enables the application developer to specify demands regarding protection from interferences. This is the third in a series of four top-level demand classes that constitute the application language.

When multiple applications share the resources of an execution platform, so-called interferences can occur. An interference is a special type of failure scenario, which is characterized by the following cause-effect chain: At the beginning of an interference, an application uses a shared platform resource, typically in an erroneous manner (e.g., it uses it for too long or modifies it in the wrong way). This resource utilization affects the resource in such a way that it is unable to provide its service as demanded by another application. This other application is affected by the misbehavior of the causative application, as it perceives a failure of the platform resource, comparable to those specified in subsection 4.5.1. Via this additional failure propagation channel, applications can interfere with each other even if there is no functional dependency between the corresponding applications.

The possibility of an interference with no functional dependency is one reason why interferences are hard to control by an application. An application is per se unaware whether a resource is shared or not, since in an integrated architecture, the deployment is specified only after the application has been developed. Therefore, the application does not know what kind of interference protection is required or if protection is required at all. The other reason why interferences are hard to handle for an application is that interferences directly affect the infrastructural resources provided by the platform. In many cases, the application depends on these resources to perform the most basic functionality (think of the CPU or the main memory). A failure of such an infrastructural resource leaves the application very badly equipped to deal with abnormal situations. Therefore, interferences are typically handled by the platform, which in most cases even prevents them.

Unhandled interferences are especially severe in a mixed-critical system. A system is called mixed-critical if the same platform hosts applications with different criticality/integrity levels. In such a scenario, interferences open channels for a failure of a low-critical application to propagate to a high-critical application, and potentially cause the high-critical application to fail. To prevent such a scenario, most safety standards demand that all applications running on the same platform are developed according to the highest integrity level amongst them if the platform is unable to prevent or control interferences.

To demand protection from such an interference, the application developer specifies a resource protection demand. Specification of a

resource protection demand always includes specification of the resource to be protected and the *critical failure modes* of the resource from the perspective of the demanding application. A protection demand is fulfilled if the platform is able to guarantee that all applications with lower criticality than the demanding application are unable to cause one of the critical failures of the corresponding resource. Which failures are critical differs between applications and has to be specified by the application developer. By specifying the failure modes that are actually critical, the application developer facilitates mediation, as the platform must only protect the resource regarding this limited number of failures.

In the literature, most authors differentiate between temporal and spatial interferences. Temporal interferences can be encountered when using a time-partitioned resource like the ECU. If overutilization of this resource by a resource user occurs, other users of the resource are affected by the resulting drop in the resource's performance. Spatial interferences, on the other hand, are encountered with space-partitioned resources like memory. A spatial interference occurs when one resource user manipulates a segment of a space-partitioned resource belonging to another user. However, there are resources like memory that have space- and time-partitioned aspects. Memory, for example, is space-partitioned regarding its separation into address regions, where a range of addresses represent a so-called memory segment that can belong to a single program or a restricted set of programs. On a system with different concurrent memory users, such as in a multi-core system, memory is also time-partitioned, since usually only one core is able to access a memory module at a time.

The VerSal language differentiates between two kinds of resource protection demands: protection demands for the basic execution resources (memory and CPU) and protection demands for the standard platform services, comparable to platform service demands. The relatively coarse-grained failure classification of temporal and spatial failures is only used for basic execution protection demands, whereas we use the common failure model specified in section 4.4.2 for the service protection demands to provide a more fine-grained differentiation of failure modes. Figure 55 shows the principal structure of the resource protection meta-model.

Regarding the integration of resource protection demands into the architecture model of the application, we have to again differentiate between basic execution resource protection demands and service protection demands. Since memory and CPU utilization is not explicitly modeled using service needs, a memory protection demand is contained in the ASWC's memory section that must be protected from interferences and a CPU protection demand references the runnable requiring protection. Other than that, the related architecture elements

of the other protection demands are the same as specified for platform service demands in Table 7 of section 4.5.1.

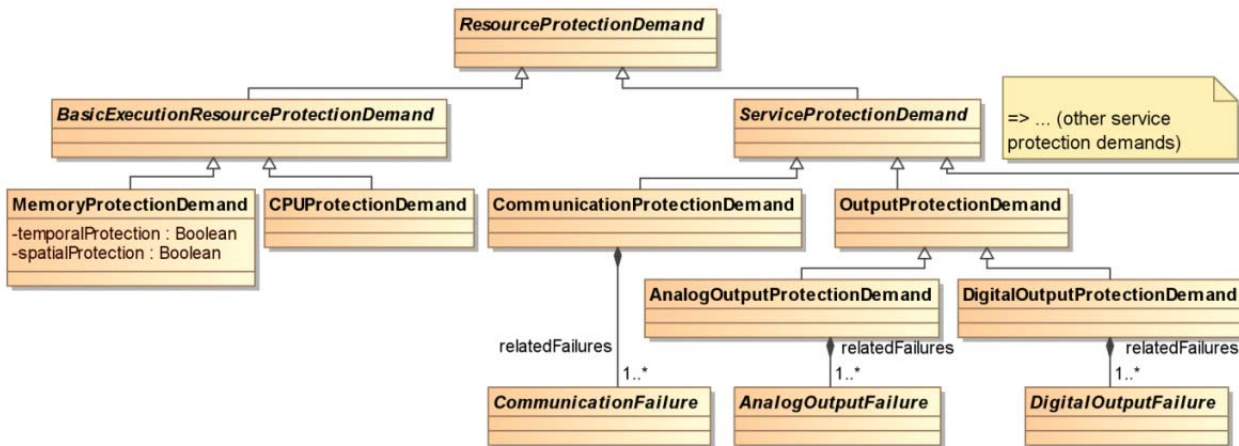


Figure 55: An excerpt of the resource protection meta-model

To reduce redundant work for the application developer, the VerSal language allows for an automatic protection demand specification in addition to manual protection demand specification. In case of a manual specification, the application developer has to specify which resource utilization requires protection and – a most tedious task – which failure modes are critical. In case automatic specification is configured, the VerSal mediator will automatically generate a resource protection demand for every service utilization including all critical failures. To do so, the automatic service uses the already specified platform service failure demands. The assumption underlying automatic specification is that if a service failure is critical regarding platform failures, then it should be critical regarding interference-related causes as well.

In the following, we will introduce some example resource protection demands based on the running example introduced in section 4.1.

Example D9 introduces a typical but also simple protection demand. The demand requires the protection from temporal interferences of the executable `v_controller_main` via the shared resource CPU.

Example D9: The executable `v_controller_main` must be protected from temporal CPU interferences (ASIL C).

Example D10 introduces a more complex protection demand owned by the software component monitoring, which uses the event service provided by the host platform. The requirement demands the protection from interferences via this shared event service. The specification includes a list identifying two failure modes that are not allowed to occur as a result of an interference.

Example D10: The monitoring.error_event service need must be protected from interferences that cause the failure modes Event Signal Commission, Event Timeout Failure (ASIL C).

4.5.4 Service Diversity Demands

A service diversity demand enables the application developer to specify demands regarding the diverse design and implementation of platform services. This is the final demand class in a series of four top-level demand classes that constitute the application language.

According to [76], Design diversity is a defense against “common mode” or “common cause” development errors in safety critical systems. It is a system design concept that attempts to reduce the possibility that the failure stemming from a common development error in one functional failure path will result in another functional failure path. This is accomplished by designing a functional failure path to be sufficiently different to minimize the likelihood that the error will manifest itself in another functional failure path implementing the system function and, then, allow an unacceptable failure event.

The concept of diversity is used in many safety standards like DO-178C [60], IEC 61508[59], and [46], although the terminology differs as some standards use the term dissimilarity or independence to describe the same or a comparable concept. Recapitulating the above definition, diversity is used so as to reduce the likelihood of common-cause systematic failures in redundant components. Depending on the safety standard, there are different demands that have to be fulfilled before diversity can be assumed. Typically, safety standards ask at least for diverse design of the relevant components and for independence between the teams developing them. Please note that diverse design and implementation are no guarantee for the freedom from common cause systematic failures, but most certification authorities accept diversity as a measure to support the corresponding claim.

In a federated architecture, diversity usually means that there is a redundant functional architecture, where the redundant channels have been implemented on the technical level using different platforms. However, in an integrated architecture, it is also possible that the redundant channels are deployed to the same platform. To support such an architecture, services offered by the platform, like two input channels, are sometimes designed and implemented diversely.

Overall, there are mainly two use cases for having diverse services on an execution platform. First, diverse services can be combined into one virtual service by the platform developer to provide a more reliable service. This design pattern, however, is invisible to the application, since

the application only sees the emerging virtual reliable service. The second use case applies, if the application is designed according to a redundant architecture (e.g., 1 out of 2) and both channels of the redundant application are deployed to the same platform. In this case, the services used by the redundant channels must be developed diversely in order not to introduce systematic common-cause failures via the platform. To support this use case, the VerSal language allows the application developer to specify service diversity demands.

If service diversity demands are specified in a comprehensive manner, a diversity demand is only fulfilled if the corresponding services fail independently with regard to every failure mode of the relevant service type. If we regard the communication service as an example, the VerSal language differentiates between five failure modes as specified in subsection 4.4.2 Communication Failure Model. If an application developer demands general diversity between two communication channels, the demand can only be fulfilled if the corresponding channels fail independently with regard to all five failure modes.

To specify diversity demands individually, the VerSal language offers failure-mode precise diversity demand specification. In this case, the application developer first identifies the critical failure modes of a service utilization, and then demands service diversity only regarding critical failure modes. If, for example, the omission failure of a communication channel was the only critical failure mode in a specific use case, it would not matter if there were systematic common causes for message corruptions on both diverse communication channels. Comparable to the automatic specification of resource protection demands, the application's demands regarding platform service failures (see section 4.5.1) can be used to identify critical failure modes automatically and specify the diversity demand accordingly.

In the VerSal language, there are three kinds of service diversity demands: input service diversity demands to demand the independence of input services, communication service diversity demands to demand the independence of communication links, and output service diversity demands to demand the independence of output services. We allow specifying diversity demands for only these service types since these are, to the best of our knowledge, the only types that are usually developed diversely on an execution platform.

When specifying a diversity demand, the application developer has to specify two channels that have to be developed diversely. Of course, the application developer is unable to directly specify a channel since the channel is a platform element. As a consequence, the developer specifies the corresponding ports of the application to demand in a transitive manner that the channels the ports are going to be deployed to are developed diversely. The VerSal language only allows the specification of

two-channel diversity since it is uncommon to have more than two diversely developed services on one platform.

In addition to the diverse channels, the application developer has to specify the failure modes of the channels that need to be independent. The application developer specifies this by picking a set of relevant failure modes from the appropriate failure model presented in section 4.4.2. If the application developer specifies a set of failure modes, every combination of failure modes must be independent. As a point in case, assume that there is a diversity demand that refers to the channels **ch1** and **ch2** and contains the failure modes **fm1** and **fm2**. In this case, all of the following pairs of failure modes must be independent to fulfill the demand: $(ch1.fm1, ch2.fm1)$, $(ch1.fm1, ch2.fm2)$, $(ch1.fm2, ch2.fm1)$ $(ch1.fm2, ch2.fm2)$.

Finally, all service diversity demands are contained in the application itself since the demand usually spans several ASWCs, leaving only the application itself as a suitable container. Figure 56 shows the resulting service diversity meta-model.

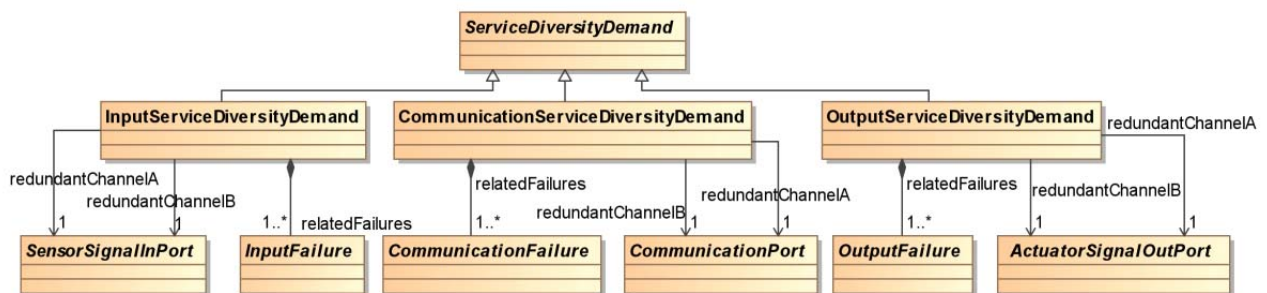


Figure 56: The service diversity meta-model

In the following, we will introduce some example service diversity demands based on the running example introduced in section 4.1.

The running example application contains two redundant sensor software components that read redundant vehicle velocity sensor values. Example D11 specifies a service diversity demand for the redundant sensor signals **v_raw_A** and **v_raw_B**. If this demand is fulfilled, the application developer can design an application safety case based on the assumption that the specified common cause failures are avoided by the platform design.

*Example D11: The analog input channels used to read the input signals **v_raw_A** and **v_raw_B** must be developed diversely. The following common-cause failure shall be avoided by means of diverse design: Analog Input Value Failure (ASIL C).*

4.6 Platform Language

In this section, we will introduce the platform language. The platform developer uses the platform language for the specification of safety-related guarantees regarding the behavior of the platform. These guarantees can be used for the mediation of demands specified using the VerSal application language introduced in section 4.5. Just like the application language, the specification of the platform language is based on the common types, attributes, and relations specified by the common language introduced in sections 4.3 and 4.4, and to some extent on the platform model, which is presented in Annex A.3.

Prior to the description of the platform language we want to note that the overall design of the platform language is in many aspects comparable to the design of the application language, but also differs significantly in other aspects. In order to avoid repeating things that were already specified in the previous section, we will refer the reader to the specification of the application language where appropriate. However, this is not always possible in order to keep up the flow of reading.

The platform developer uses the platform language to specify the vertical safety interface of the platform. The platform safety interface contains all the guarantees regarding the behavior of the platform that can be used to fulfill the demands of the platform's guest applications. In order to provide a sufficient level of trustworthiness, the platform development process must include steps that provide evidence about the reliability of the guarantees. In certain industries, it might also be necessary for the platform to undergo assessment or certification; the guarantees can then be used to support the safety case of the guest applications.

The structure of the application and platform language is based on the observation that there are four major classes of safety-related dependencies between an application and a platform. Analogously to the application language, the platform language consists of four basic types of guarantees: platform service guarantees, health monitoring guarantees, resource protection guarantees, and service diversity guarantees.

With *platform service guarantees*, the platform developer specifies the platform's capabilities of providing reliable infrastructural services and detecting failures should they occur. *Health monitoring guarantees* enable the platform developer to specify the platform's mechanisms for detecting application failures and executing failure control reactions. *Resource protection guarantees*, on the other hand, are used to specify the platform's capability of protecting resources from interferences. Finally, using *service diversity guarantees*, the platform developer

specifies which platform services are developed diversely and can therefore be used in various redundancy safety concepts.

Figure 57 shows the top-level structure of the platform language meta-model including the different demand classes. If you compare this figure with Figure 51 depicting the structure of the application language, you will see the analog structure of both models, which simplifies mediation. Just as in the application language, guarantees are also not contained in the vertical platform interface, but in the corresponding service or element that is mainly responsible for providing the guarantee. The containment relations of the different guarantees will be introduced in the following subsection.

Regarding the parameterization of demands, failure modes, and failure reactions, the semantics of the platform language is completely different than the semantics of the application language. In the case of the application language, parameters are used to specify that single relevant failure mode or reaction as precisely as possible. The platform developer, however, does not want to specify one failure mode or reaction, but the set or the range of failure modes/reactions that the platform is capable of handling. To allow this, specifying parameters always spans a range of possible failure modes or reactions. For instance, if a platform developer specifies a failure detection time of x ms, this means that all failure modes with a detection time equal to or longer than x ms can be handled.

Platform service guarantees will be introduced in subsection 4.6.1, health monitoring guarantees in subsection 4.6.2, resource protection guarantees in subsection 4.6.3, and finally service diversity guarantees in subsection 4.6.4.

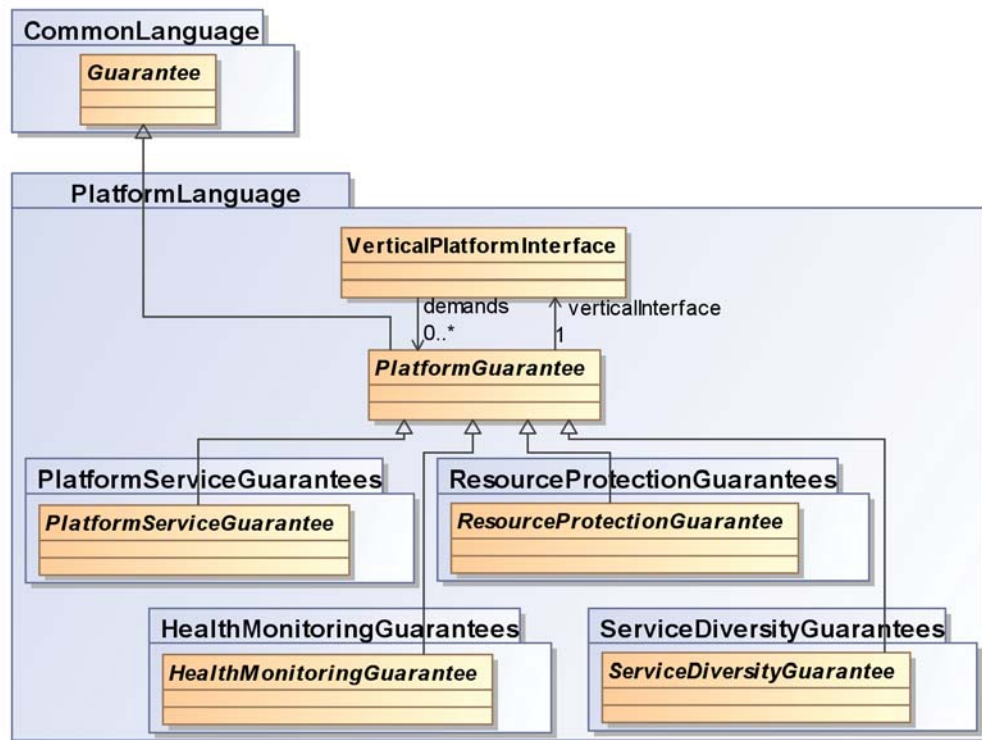


Figure 57: The top-level meta-model of the platform language

4.6.1 Platform Service Guarantees

A platform service failure guarantee enables the platform developer to specify guarantees regarding the platform's capabilities of avoiding or detecting failures of its provided services. This is the first in a series of four top-level guarantee classes that constitute the platform language.

In order for a platform to be used in a safety-critical system, it must provide safe services to enable an application to provide safe functions. Safety is typically achieved by one of the following means [77]:

A) **Fault tolerance** is intended to preserve the delivery of safe service in the presence of active faults. Fault tolerance is a design measure that usually involves the development of mechanisms for performing error detection and error handling. Regarding platform services, this means that the platform must be able to detect erroneous services and allow for appropriate error handling (the latter aspect is covered in the next section). Error handling does not necessarily imply that the service user perceives the service as failed. If the platform is able to perform error handling in a fail-operation manner (e.g., using rollback or redundancy), the service users might be able to perform their tasks uninterrupted.

B) **Fault avoidance** or *fault prevention* is attained by quality control techniques employed during the design and manufacturing of hardware and software. They include structured programming, information hiding,

modularization, etc., for software, and rigorous design rules for hardware.

*C) **Fault removal** is performed both during the development phase, and during the operational life of a system. Fault removal during the development phase of a system life-cycle consists of three steps: verification, diagnosis, correction. Verification is the process of checking whether the system adheres to given properties, termed the verification conditions. If it does not, the other two steps follow: diagnosing the fault(s) that prevented the verification conditions from being fulfilled, and then performing the necessary corrections. In other words, fault removal includes verification and validation measures like testing, inspection, static analyses, model checking, etc.*

Just like platform service demands, platform guarantees are divided into detection and avoidance guarantees (for more information regarding the semantics of detection and avoidance, please refer to section 4.5.1). The availability of a platform service guarantee, be it detection or avoidance, is mainly determined by the implemented fault tolerance mechanisms. The integrity level of the provided guarantee, however, is determined by the fault avoidance and fault removal steps that the platform developer performed during the platform development process. Which kind of development process is sufficient to claim a certain level of integrity is usually specified by the applicable safety standard. On the other hand, determining whether a certain composition of mechanisms is sufficient to claim the detection or avoidance of a failure must be argued by the platform developer on a case by case basis²⁷.

The steps involved in developing a certain behavior that is backed by evidence and compliant with a safety standard is relevant for every kind of execution platform. However, if the execution platform is part of an integrated system it is always developed independent of the guest applications. Consequently, the platform guarantees have to be chosen before the applications and their demands are known. In order to pick the right guarantees, the platform developer has to rely on experience or on standardized safety concept patterns in the relevant industrial domain. If the platform is developed for the automotive industry, it should be capable of hosting an application that is safeguarded using the standardized E-Gas monitoring concept.

We will now continue with a description of the modeling of platform service guarantees, including additional parameters, the integration of the demands into the architecture model of the application, and the usage of the common platform service failure model.

²⁷ There are some standards, like [59] and [46], that provide guidelines for the choice of mechanisms that are appropriate for handling certain failure modes.

On the top level, platform service guarantees are separated into guarantees for failure detection (**PlatformServiceFailureDetectionGuarantee**) and for failure avoidance (**PlatformServiceFailureAvoidanceGuarantee**). Those abstract guarantees are further differentiated into failure- or element-specific guarantees, like communication failure detection and avoidance guarantees. The platform developer instantiates these guarantees to specify the safety-related behavior of the platform. To configure the guarantee, the platform developer chooses the related platform element of the corresponding type (e.g., a communication link for communication failure guarantees) and a failure from the matching failure model (e.g., the corruption failure from the communication failure mode). The related element and the related failure mode are specified using containment relations. The guarantee is contained in the related platform element and contains the related failure mode.

Regarding the parameterization of the guarantee, the platform developer has to specify the integrity level of the guarantee (a parameter inherited from the abstract interface requirement class, see section 4.3.4 Integrity Level), the failure detection time of the guarantee in case it is a detection guarantee (see section 4.5.1), and all the parameters of the related failure-mode (see section 4.4.2 for the failure modes of each parameter). The platform developer uses the parameters to describe the range of detectable or avoidable failures. Specifying a failure detection time of *x ms* means that failures with a failure detection time of *x ms* or larger can be detected.

Figure 58 shows an excerpt of the platform service guarantee meta-model.

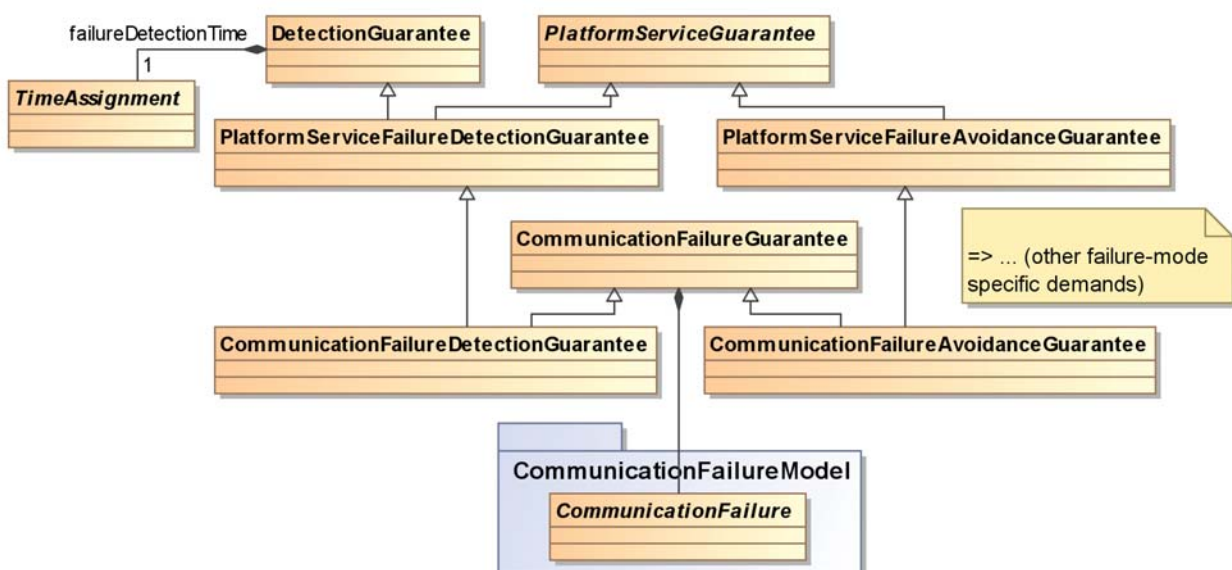


Figure 58: The top-level structure of the platform service guarantee meta-model

Before we provide a list of all available platform service guarantees, an explanation of a particularity of the platform model is in order. There are certain platform elements called platform objects that are generated by the integrator during platform configuration, as opposed to every other kind of platform element (e.g., an input channel) that is modeled by the platform developer during platform development. Every platform object is a logical software object that can only be generated when the needs of the host applications are known. Example platform objects are software timers, semaphores, mutexes, events, partitions, etc.

Since these elements are only generated during integration, the platform developer cannot directly specify guarantees for them. Instead, the platform developer specifies a platform object guarantee template containing the guarantees of the platform object. This guarantee template is contained in the platform element or platform service that provides the platform object. As an example, the mutex guarantee template is contained in the mutex service. The VerSal mediator will automatically generate these object-related guarantees when the integrator instantiates a platform object (see section 5.1.1).

As a consequence of this procedure, object-related guarantee types can be contained in the service providing the platform object, and in the platform object itself. Nevertheless, the object-related guarantees contained in the service play no role in mediation. As soon as the integration phase is finished, the mediation algorithm will only use the guarantees generated for the platform object to fulfill the application guarantees.

Table 8 shows all available platform service guarantees, including their related platform elements and their related failure modes. Platform object related platform service guarantees are distinguished by having two related platform elements, the platform object container and the platform object itself.

Table 8: A list of all platform service demands including their related application elements and failure modes.

<i>Guarantee</i>	<i>Related platform element</i>	<i>Related failure mode</i>	<i>Failure model</i>
MutexFailure Guarantee	MutexService Mutex	MutexFailure	Synchronization-FailureModel
EventFailure Guarantee	EventService Event	EventFailure	“
Communication FailureGuarantee	Communication Link	Communication-Failure	Communication-FailureModel
DigitalInput	DigitalInput	DigitalInput-	InputFailure-

FailureGuarantee	Channel	Failure	Model
AnalogInput FailureGuarantee	AnalogInput Channel	AnalogInput- Failure	“
DigitalOutput FailureGuarantee	DigitalOutput Channel	DigitalOutput- Failure	OutputFailure- Model
AnalogOutput FailureGuarantee	AnalogOutput Channel	AnalogOutput- Failure	“
GlobalTimeFailure Guarantee	GlobalTime- Service	GlobalTime- Failure	TimeService- FailureModel
RelativeTime FailureGuarantee	TimerService Timer	RelativeTime- Failure	“
WaitTimeFailure Guarantee	WaitService	WaitTime-Failure	“
MemoryService FailureGuarantee	MemoryService File	MemoryService- Failure	MemoryService- FailureModel
TTRunnable- Scheduling FailureGuarantee	TimeTriggered Task	TTRunnable- Scheduling- Failure	Scheduling- FailureModel
RunnableScheduling FailureGuarantee	Task	GeneralRunnable- Scheduling- Failure	“
Interrupt- Scheduling- FailureGuarantee	Interrupt	Interrupt- Scheduling- Failure	“
CoreRelatedFailure Guarantee	Core	CoreRelated- Failure	BasicExecution- FailureModel
MainMemoryFailure Guarantee	MemoryModule MemorySegment	MainMemory- Failure	“
PowerSupplyFailure Guarantee	Platform	PowerSupply- Failure	“

In the following, we will introduce some example platform service guarantees based on the running example introduced in section 4.1.

The platform service guarantee Example G1 specifies a guarantee provided for the ADC input channel called **voltage_in**. Please note that the input channel comprises hardware as well as software components, such as drivers provided by the platform.

Example G1: A sampling latency of input signals received via voltage_in larger than 0.1ms can be avoided (ASIL B).

Example G2 specifies a guarantee provided by the **can0** communication link. Comparable to the previous guarantee, **can0** comprises hardware (controller and transceiver) as well the communication software stack provided by the platform.

*Example G2: Corruptions of messages transmitted via communication link **can0** can be detected (ASIL C).*

4.6.2 Health Monitoring Guarantees

A health monitoring guarantee enables the platform developer to specify the platform's capabilities of detecting application failures and executing failure recovery reactions. This is the second in a series of four top-level guarantee classes that constitute the platform language.

Both mechanisms, failure detection and recovery, can be used to improve the fault tolerance of an application. With an application monitoring mechanism, the platform is able to detect erroneous application behavior, and by providing a failure recovery mechanism, the platform is even able to help the application recover from the situation. Additionally, failure recovery mechanisms can be used by the platform to recover from platform service failures (see previous section for additional information regarding platform service failure guarantees).

Comparable to the health monitoring demand model, the corresponding guarantee model is divided into two parts. The first part contains guarantees regarding application monitoring and failure detection and the second part contains guarantees regarding failure recovery mechanisms. Figure 59 gives an overview of the health monitoring model. This section shares the split structure of the model: Application monitoring guarantees are introduced in the next subsection and failure reaction guarantees are introduced after that.

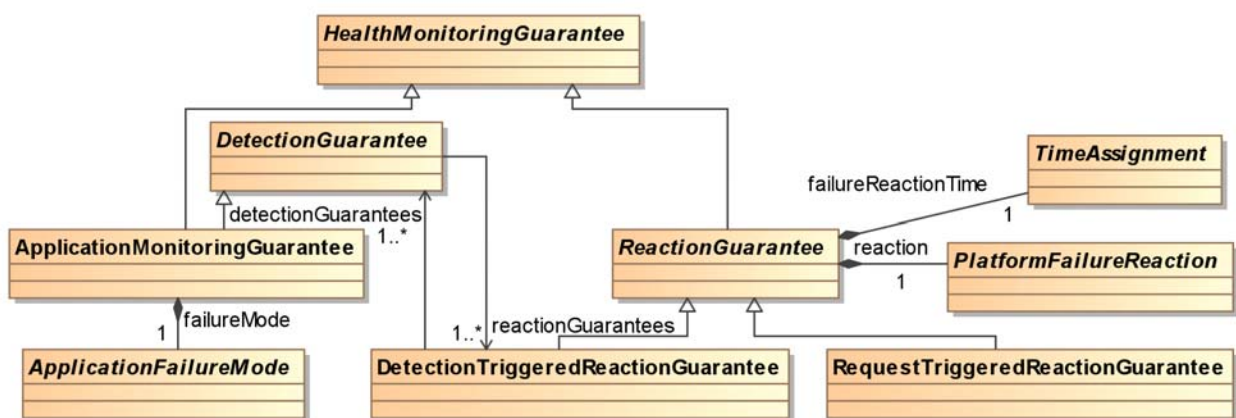


Figure 59: The health monitoring guarantee meta-model

Application Monitoring Guarantees

The platform developer uses application monitoring guarantees to specify the platform's capability of monitoring the application behavior and detecting deviations from the application's specified behavior, i.e., application failures. For the specification of platform-detectable application failures, the model references the common application failure model presented in section 4.4.3.

Since in an integrated architecture the guest applications are unknown during a platform's development, the applications' nominal behavior is not known either. Therefore, application monitoring mechanisms have to be configurable to enable the system integrator to adapt the mechanisms to the specific behavior of the guest applications. Consequently, the platform developer is only able to guarantee the correct functioning of the monitoring mechanisms under the premise that the mechanisms are configured correctly. The platform developer models this premise by specifying a conditional application monitoring guarantee.

An application monitoring guarantee inherits the possibility of containing conditions from the general interface requirement class as specified in section 4.3.2. At first glance, there are two types of conditions that are qualified for specifying this kind of premise: configuration-dependent conditions and manual conditions. However, upon closer examination, configuration-dependent conditions are not suitable for automatically checking for the correctness of a monitoring mechanism's configuration. A configuration-dependent condition cannot specify the required value of a platform configuration parameter if the value depends on a parameter of the application. To allow such a construct, all configuration parameters of the application must be known during platform development. This is not the case in the VerSal language. If we adapt the VerSal language to a predefined standard like AUTOSAR, where the application configuration parameters are fixed, such a mechanism would be possible. For now, the platform developer has to specify a manual condition to require the integrator to check the configuration for correctness.

If the platform developer specifies an application monitoring guarantee, the equally named class (see Figure 59) must be instantiated. Following the containment rule of the platform language, all application monitoring guarantees are contained in the platform's application monitoring service, since this service is responsible for providing monitoring mechanisms. When the demand is instantiated, the platform developer chooses the application failure that the monitoring facility can detect from the common application failure model specified in section 4.4.3. If the application monitoring facility is capable of detecting more

than one application failure, the platform developer has to specify several application monitoring demands.

As always, the next step in specifying the guarantees is to parameterize the guarantee itself as well as the related common element, in this case the related application failure. Since every application monitoring guarantee is a detection guarantee, this includes the minimum failure detection time. With regard to the parameterization of the application failure mode, the developer has to parameterize the failure-mode-specific parameters and the supervised entity or, in the case of the platform, the supervisable entities. The VerSal language provides the developer with three choices: All executable entities are supervisable (ISR and runnable), only runnables are supervisable, or only ISRs are supervisable. To specify that no supervision is possible, the platform developer does not specify the corresponding guarantee.

In the following, we will introduce some example application monitoring guarantees based on the running example introduced in section 4.1.

Example G3 specifies the capabilities of the platform with respect to detection execution time failures. As specified above, the platform developer has to specify which types of executable entities can be supervised.

Example G3: The platform is capable of detecting execution time failures of runnables (ASIL C).

Example G4 specifies a guarantee with respect to the detection of logical sequence failures. Furthermore, example G4 specifies a manual condition which states that the guarantee is only valid if the monitoring facility is configured appropriately. The manual condition further demands the generation of an evidence. Since this condition is a manual condition, the integrator has to manually set the status of the condition before the mediator is capable of using the specified guarantee. For more information regarding the specification of manual conditions and conditions in general, please refer to section 4.3.2.

Example G4: The platform is capable of detecting logical sequence failures of runnables and ISRs (ASIL C). Conditions apply: "The monitoring facility must be configured appropriately". Corresponding evidences must be generated: "Review of configuration file".

Failure Reaction Guarantees

The platform developer uses failure reaction guarantees to specify the platform's capability of performing reactions that help application to recover from application or platform service failures. For the specification

of recovery reactions, the guarantee model references the common failure reactions presented in section 4.4.4.

The failure recovery reactions specified in section 4.4.4 are common reactions found in many platforms. Comparable to application monitoring facilities, some of the reactions are configurable to allow adapting the reaction to the specific needs of the guest application. If that is the case, the platform developer must specify a conditioned guarantee in order to model that the corresponding failure reaction can only be provided correctly under the premise that it is configured correctly. For more information regarding conditional guarantees for configurable platform mechanisms, please refer to section 4.3.2.

There are two kinds of failure reaction guarantees: request-triggered reaction guarantees and detection-triggered reaction guarantees. The platform developer models a request-triggered reaction guarantee to specify that the platform is capable of performing a certain recovery reaction on application request, i.e., via an API call. In contrast, the platform developer models a detection-triggered reaction guarantee to specify that the platform is capable of performing a recovery reaction as an automatic result of a detected failure.

To model both kinds of guarantees, the platform developer must specify the guarantee's failure reaction time and related failure reaction. The failure reaction time is the maximum time between the triggering of the reaction, be that trigger a request via the API or a failure detection, and the time when the execution of the reaction has finished. The related failure reaction is chosen from the failure recovery model specified in the previous section. Some of these failure reactions have parameters that have to be set when the platform developer specifies a failure reaction guarantee. When inspecting the failure model you will realize that some reactions reference multiple parameters of the same type. The "issue default signal" reaction can, for instance, contain several default signals. For a reaction demand specified by the application this makes no sense, which is why only the first default signal of the list is relevant in case the reaction is specified in the context of a demand. For a reaction guarantee, however, the platform developer is able to specify all default signals the platform is able to send. In case there is no limitation on the default signals, the platform developer sets the default signal list to NULL. In case the platform is unable to issue default signals at all, the platform developer specifies no such guarantee.

This pattern is also used to specify the architecture element of a failure recovery reaction affected by a reaction. This kind of reference identifies, for example, which partition can be shut down by a shutdown reaction or which channel a default signal can be sent on (for more information, refer to section 4.4.4). When used to specify a platform reaction guarantee, the platform developer uses these references to specify all

possible reaction-affected architecture elements, for example, all output channels that are able to issue a default signal. Table 9 shows a list of all failure reactions and how their reaction-affected architecture reference can be parameterized when the failure recovery reaction is used in the context of a reaction guarantee.

Table 9: A list of all platform service demands including their related application elements and failure modes.

<i>Reaction</i>	<i>Reaction-affected platform element</i>
RestartTask	All Task Objects, Task
ShutDownTask	“
RestartPartition	All Partition Objects, Partition
ShutDownPartition	“
RestartPlatform	ExecutionPlatform
ShutDownPlatform	ExecutionPlatform
SendDefaultMessage	All ComLinks, ComLink
IssueAnalogDefault-Signal	All AnalogOutputChannels, AnalogOutputChannel
IssueDigitalDefault-Signal	All DigitalOutputChannels, DigitalOutputChannel
Indication	All ASWCs
HandlerExecution	All Runnables

There are two more important aspects regarding the specification of reaction-affected platform elements: Most of the references are conditional, and there are wildcard references.

Using conditional references, the platform developer is able to specify that a reaction can be executed on a specific element under the premise that a certain condition is fulfilled. It is, for example, possible to specify that the related element must be configured appropriately (e.g., a task must have the property `restart_enabled` set to `true` before it can be restarted) or the integrator must perform a manual check before the element can be used (e.g., a partition may only be shut down if it has been checked that it does not affect another application).

Wildcard references like “all runnables” allow the platform developer to specify that a certain reaction can be performed on all elements of a type. Conditional references can also be used together with platform object wildcard references (partitions and tasks). If the platform

developer specifies such a reference, this means that every platform object of that type is a potential reaction-affected element if the specified condition holds.

The parameters described above have to be set for both kinds of reaction guarantees, detection-triggered guarantees and request-triggered guarantees. Thus, when modeling a detection-triggered guarantee, the platform developer has to additionally specify which failure detections are able to trigger the corresponding reaction. The platform developer selects these failure detections via the list of specified detection guarantees, which includes platform service detection guarantees and application monitoring guarantees.

In the following, we will introduce some examples of the rather complex specification of failure reaction guarantees. The example guarantees are based on the running example introduced in section 4.1.

Example G5 specifies a guarantee that states that the platform is capable of restarting tasks upon request. Since the guarantee is a request-triggered reaction, the possible detection triggers do not have to be specified. However, the possible targets of the restart have to be specified by the platform developer. For simplicity's sake, we chose to use the wildcard "**all tasks**" for this guarantee, meaning that in principle, every task can be restarted. However, a manual condition applies, which states that the caller must have sufficient rights to restart the task. Since this is a manual condition, the integrator would have to check whether this is true for every user of this guarantee.

Example G5: The platform is capable of restarting tasks upon request. Possible restart targets: all tasks. Conditions apply: "The caller must have sufficient rights to request the restart" (ASIL C).

The second guarantee provided by example G6 is a detection-triggered reaction guarantee. Consequently, the possible detection triggers have to be specified. We use the wildcard "**all detections**" to specify that every detection mechanism is capable of triggering this "shutdown partition" reaction. With respect to the possible target partitions, there is a condition that limits the targets to partitions that have the parameter **restart_enabled** set to **true**. Additionally, the same manual condition that applied for example G5 applies for example G6 as well.

Example G6: The platform is capable of shutting down partitions upon detection. Possible triggers: all failure detection events. Possible restart targets: partition objects with the configuration condition "restart_enabled == true". Conditions apply: "The caller must have sufficient rights to request the shutdown" (ASIL C).

4.6.3 Resource Protection Guarantees

A resource protection guarantee enables the platform developer to specify the platform's capabilities of protecting the application from interferences. This is the third in a series of four top-level guarantee classes that constitute the platform language.

An interference failure is rather a failure scenario than an actual failure mode. In an interference scenario, an application is affected by a failure of a platform resource, but this failure is not caused by the resource provider, i.e., the platform, but by another resource user, i.e., another application. Most safety standards demand that all applications that can interfere with each other must be developed according to the same integrity level. Hence, if the platform does not protect applications from interferences, this will either lead to deployment incompatibilities (if the applications cannot be developed according to a higher integrity level) or to an increase in development costs (if the applications are developed according to a higher level). More information regarding interferences is found in section 4.5.3.

If the platform developer uses a resource protection guarantee to specify the platform's capability to offer protection from interference, the platform developer has to choose from one of thirteen different protection guarantee classes. Each class represents a guarantee to protect a different type of resource, including cores, memory modules, I/O- and com-channels, and the different services provided by a platform. Since it is not always possible to provide comprehensive protection, the platform developer has to additionally specify for each instantiated guarantee which failure modes are protected from interference causes. For most of the protection guarantee types, the platform developer uses the detailed resource-specific failure models introduced in section 4.4.2. However, for the specification of CPU and memory protection guarantees, there is no refined failure model, which is why the developer has to use the coarse-grained temporal/spatial interference differentiation²⁸²⁹. Figure 60 provides an overview of the resource protection guarantee meta-model.

²⁸ On the abstraction level applicable to the VerSAI approach, a CPU poses only temporal interference failures.

²⁹ The difference between temporal and spatial interferences is described in section 4.5.3.

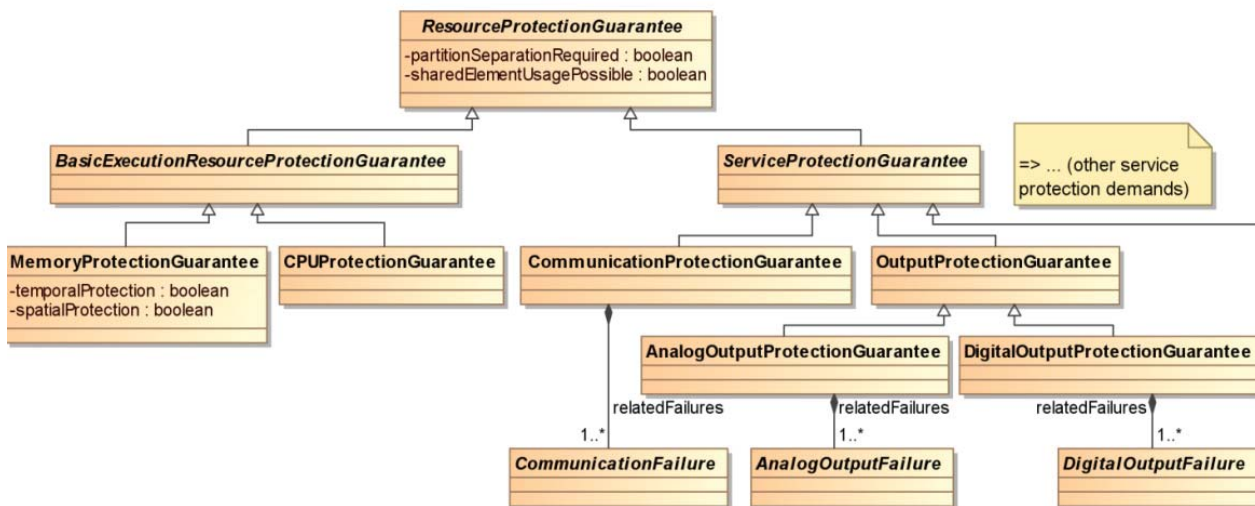


Figure 60: An overview of the resource protection guarantee model

We have already argued that comprehensive protection guarantees are hard to achieve; this is also true regarding aspects other than the failure modes to protect against. Therefore, the VerSal language offers more gradations for precisely specifying a platform's protection capabilities. First, many platforms offer so-called partitions. A partition is a logical grouping of ASWCs, where ASWCs in different partitions are not able to interfere with each other. Or put in other words, protection is only provided between ASWCs that reside in different partitions. If that is the case, the platform developer must set the **partitionSeparationRequired** parameter of the corresponding guarantee to **true**. The other gradation aspect regards the shared usage of the protected resource. It is much easier for the platform to provide protection from other users if the users are not allowed to access the resource at all. However, in case protection is required for users that share a resource, more sophisticated protection mechanisms are required. To specify that protection is available even between shared resource users, the platform developer must set the **sharedElementUsagePossible** parameter to **true**. In case the protected platform element is a platform object (e.g., a mutex), there are even two levels of resource sharing. The first is sharing the object itself, which is covered by the previously mentioned parameter. The second level is sharing access to the service that provides the platform object (e.g., the synchronization service). To specify if protection is provided for users accessing the same service, there is an additional parameter called **sharedServiceUsagePossible** for parameterizing platform-object-related protection guarantees.

Comparable to platform service guarantees, resource protection guarantees are always contained in the related platform element. In this case, this is the protected resource. A CPU protection guarantee is, for example, contained in one of the platform's CPUs/cores.

In the following, we will introduce some examples resource protection guarantees based on the running example introduced in section 4.1.

Example G7 shows a temporal CPU protection guarantee. The **partitionSeparationRequired** parameter is set to **true**, meaning that protection is only guaranteed if mixed-critical executables run in different partitions. The **sharedElementUsagePossible** parameter is set to **false**, which additionally demands that mixed-critical executables have to be mapped to a different core as well.

Example G7: The platform is capable of protecting core0 from temporal interferences. Mixed-critical users must be allocated to different partitions. Mixed-critical users must be allocated to a different CPU (ASIL C).

Example G8 specifies a protection guarantee for the **event_service** service provided by our example platform. Since an event service manages platform objects (the events), we have an additional parameter for precisely specifying the protection guarantee. In the example guarantee, the **partitionSeparationRequired** parameter is set to **false**. As a consequence, protection is still valid even if mixed-critical software components are not separated by partition boundaries. The new **sharedServiceUsagePossible** parameter is set to **true**, which means that mixed-critical software components are even allowed to share/use the service. However, the **sharedElementUsagePossible** parameter is set to **false**, which means that protection cannot be guaranteed if mixed-critical software components access the same event.

Example G8: The platform is capable of protecting the service event_service from interferences that cause the failure modes Event Signal Commission, Event Signal Omission, Event Timeout Failure. Mixed-critical users do not have to be allocated to different partitions. Mixed-critical users are allowed to use the same service. Mixed-critical users are not allowed to use the same event (ASIL C).

4.6.4 Service Diversity Guarantees

A resource protection guarantee enables the platform developer to specify guarantees regarding the availability of diversely designed and implemented platform services. This is the final guarantee in a series of four top-level guarantee classes that constitute the platform language.

The goal of diversely developed services is to reduce the probability of common-cause systematic failures in both services. Consequently, design diversity is only applied to two (or possibly more) services of the same type, i.e., two input channels. A platform developer can use diversely developed services to achieve two kinds of goals. On the one hand, a platform developer can provide a high-integrity service by combining

two diversely developed services with lower integrity. Such integrity level decomposition is done, for example, when developing systems according to the automotive safety standard ISO 26262 [46]. However, such a design is concealed from the eyes of the application developer, as the developer would only see a platform service with high integrity. Consequently, no service diversity guarantee is required in this case. On the other hand, the application developer himself could follow the same integrity level decomposition strategy and develop two functional channels as well. In this case, the application developer must ensure that there is sufficient reason for assuming that both channels have a low probability of common-cause systematic failures. If both channels use the same or two equally developed services, the possibility of a systematic failure of the service(s) subverts the independence argument of the application developer. To support diverse channels on the application level that are both deployed to the same or the same kind of platform, a platform provides diversely developed services.

Which development techniques and evidences are required before dissimilarity can be claimed differs from standard to standard. Usually the techniques evolve around: use of different compilers, linkers, and loaders; use of different programming languages; design and development performed by independent teams with restricted interactions; designs that follow different principles (e.g., current and voltage-based analog input channels).

When the platform developer specifies a service diversity guarantee, the developer chooses from three kinds of guarantees: one for modeling diverse input channels, one for modeling diverse communication links, and one for modeling diverse output channels. Other platform service types are commonly not provided in a diverse manner and the VerSal language therefore provides no such service diversity guarantees.

When modeling a diversity guarantee, the platform developer specifies two diversely developed channels; the VerSal language provides no support for three-way diversity. Furthermore, the developer specifies all failure modes of the diversely developed channels when there is sufficient evidence to support the claim that there are no common-cause systematic failures. When specifying a set of failure modes, every possible combination of two failure modes must be free from common-cause systematic failures (see section 4.5.4 for an example).

As a last point regarding the specification of diversity guarantees, all diversity guarantees are contained in the platform itself. This is because a diversity guarantee spans several channels, so that we chose the container of both channels as a container for the reference. Figure 61 provides an overview of the service diversity guarantee meta-model.

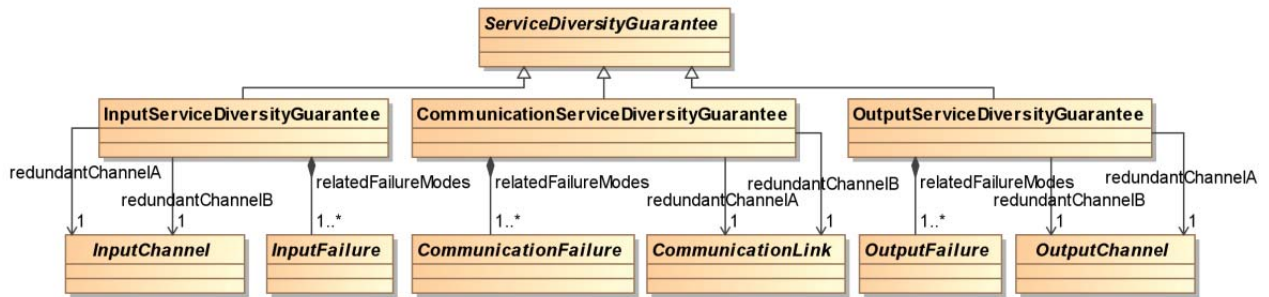


Figure 61: The service diversity guarantee meta-model

In the following, we will introduce an example service diversity guarantee based on the running example introduced in section 4.1.

Example G9 specifies a diversity guarantee that is analog to the example demand D11 specified in section 4.5.4. The guarantee states that the voltage measuring input channel **voltage_in** and the current measuring input channel **current_in** are developed diversely and that common-cause value failures are avoided.

*Example G9: The analog input channels **voltage_in** and **current_in** are designed diversely. There are no common-cause failures with respect to the following failure mode: Analog Input Value Failure (ASIL C).*

5 Interface Mediation

This chapter is the second of three chapters describing our methods for “Efficiently Deploying Safety-Critical Applications onto Open Integrated Architectures”. In this particular chapter, we will describe the realization of our mediation algorithm for checking if an application is capable of executing safely on a particular execution platform. This corresponds to the second contribution of this thesis as specified in chapter 1.

Contrib.2

Interface Mediation: *Developing an automated process for checking the safety compatibility of an application and a platform in an open integrated architecture.*

The automated mediation process provided by the VerSal method is embedded into the general task of configuration and deployment. Inputs to this phase are the previously developed applications and platforms, as well as the preliminary deployment plan. Generally speaking, the integrator integrates the applications and platforms to build the final system during configuration and deployment. With respect to the VerSal method, the integrator uses the previously specified vertical safety interfaces of applications and platforms together with the VerSal mediator to decide whether the applications are capable of executing safely on the given platforms. The configuration and deployment task is divided into several steps. Since the VerSal mediator performs automated actions in several of these steps, we will discuss the different steps in the following.

In the first step of the configuration and deployment task, the integrator configures the general-purpose execution platforms, and sometimes also the applications, so that the platform is capable of hosting the application at hand. Since the vertical safety interfaces specified by the platform developer and the application developer are integrated with the application and platform model, the configuration automatically affects the safety interfaces as well. After configuration, the integrator implements the deployment plan by mapping the components of the application onto the resources and services provided by the platform. This step also affects certain elements of the safety interface, but primarily, the mapping provides information required for the mediator. The VerSal mediator uses the deployment data to identify the guarantees that are basically capable of fulfilling the individual demands and, in a next step, for checking whether the identified guarantees are capable of fulfilling the demand. After the mediator has performed this procedure for every demand, the mediator provides the results of the mediation to the integrator. If the mediation requires further manual decision making,

the integrator is capable of performing the required actions and then retriggers the mediation. When there are no more points that require manual decision, the mediation is completed and can finally be assessed as failed or successful. An overview of the configuration and integration task is shown in Figure 62³⁰.

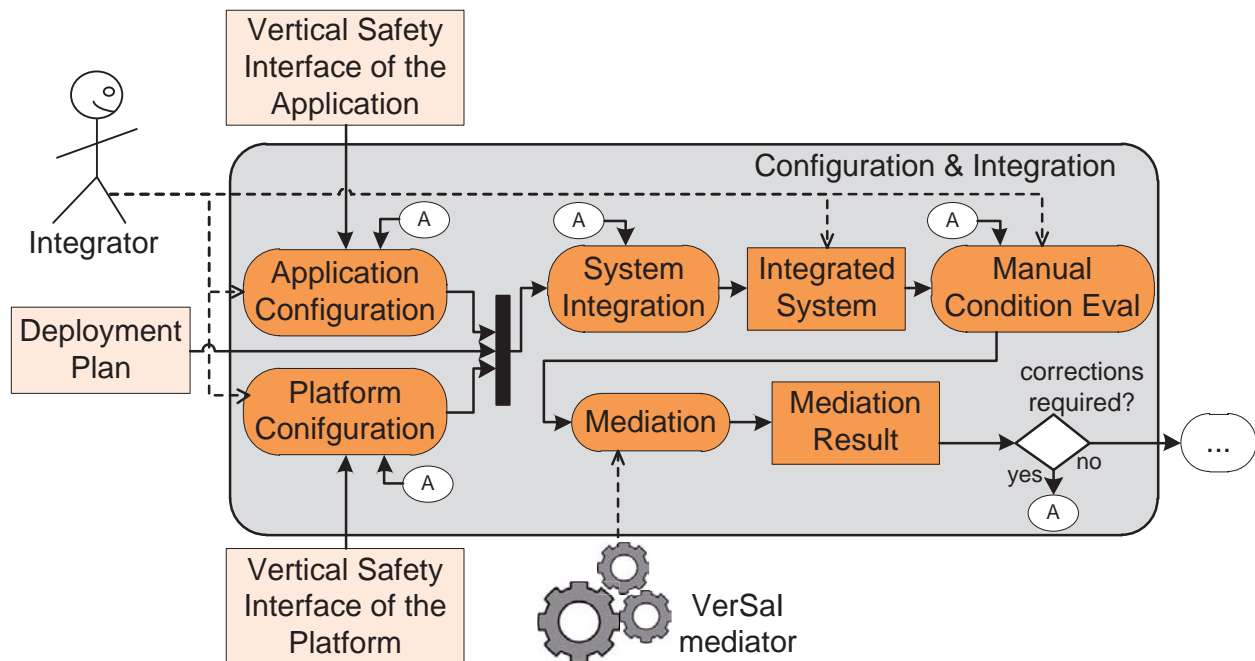


Figure 62: An overview of the VerSal mediation in the context of the overall VerSal method

In this chapter, we will introduce the afore-described process with a focus on the automatic actions that are performed by the mediator. The mediator's main task is to perform the mediation itself, but it also assists the integrator during several integration steps. The mediator performs the following tasks:

- During and after configuration: automatic specification of platform object guarantees and automatic evaluation of configuration dependent conditions.
- After system integration: automatic evaluation of deployment-dependent conditions.
- Before mediation: evaluation of manual conditions.
- Mediation: checking whether all application demands are met by the available platform guarantees.
- After mediation: visualizing the mediation results and fixing fixable issues.

³⁰ The interface between this task and the residual development process is shown in chapter 3.

This chapter is structured in accordance with these tasks. Automatic handling of platform objects and configuration-dependent conditions is explained in 5.1. The evaluation of deployment-dependent conditions is provided by section 5.2, whereas the evaluation of manual conditions is shown in section 5.3. The main focus of this chapter is the specification of the mediation algorithms in section 5.4. We conclude this chapter with the visualization of the mediation results provided by the VerSal mediator in section 5.5.

5.1 Configuration

During configuration, the integrator can adapt the applications and the platforms to their new system context. However, the integrator cannot adapt the behavior of applications and platforms freely; application and platform behavior can only be adapted regarding the variation points envisaged and designed by the developers. Besides configuration, there are other stages in the life-cycle of a system that contain variation points. However, with regard to the mediation of applications and platforms, we only consider variation points resolved during configuration.

Usually, an application is not as configurable as an execution platform. The reason is that a general-purpose execution platform draws its main value from being adjustable and able to host as many kinds of applications as possible, whereas applications are often developed for a single kind of system, leaving no need for configuration-time adaptation. Nevertheless, especially in the automotive domain, some software applications are developed in an adaptable way, so as to efficiently and flexibly use a product in different systems. Systems like the ESC or the cruise control are often developed once and then adapted for individual makes and models. This adaptation influences the internal behavior of the component, like the controller parameters, but even changes at the component's interface are possible. Regarding the cruise control, for example, most vendors have a different operation concept including different kinds of levers and buttons that can result in different software interfaces.

A different controller behavior can easily influence the parameters of an application demand, such as the demand's criticality (think, for example, of a higher maximum velocity that the application is allowed to operate at). Changes in the interface can even make certain demands obsolete, for example, when a certain input channel is not even used in a particular configuration. Modeling all these configuration dependencies is possible using the configuration-dependent conditions provided by the VerSal language and introduced in section 4.3.2.

The concept of configuration-dependent guarantees is especially valuable for specifying the vertical interface of a platform. On the one

hand, several of a platform's safety mechanisms can be deactivated for the sake of performance when using the platform in a non-safety-critical system. On the other hand, there are several safety mechanisms that must be configured appropriately before their principal capabilities of detecting or avoiding failures turn into actual guarantees.

As introduced in the preface to chapter 5, the first task of the integrator is to configure the system's applications and platforms. After the configuration is completed, all configuration-dependent conditions can be resolved. This step is automatically performed by the VerSal mediator in the transition between the configuration and the integration step. We will describe how this is done in subsection 5.1.2.

However, in addition to configuration parameters, there is another way to adapt a platform, which has to be covered by our approach. During platform configuration, the integrator creates so-called platform objects. A platform object is a logical³¹ component like a task or a software timer, which is only instantiated when the needs of an application are known, i.e., during configuration when the deployment is already planned. Since platform objects are created during configuration, the platform developer is unable to specify configuration parameters and guarantees for them. Instead, the developer specifies configuration and guarantee templates to specify the safety-related capabilities of platform objects. These templates are used to automatically instantiate configuration parameters and guarantees when the integrator instantiates a platform object. The process of automatic platform object support is introduced in section 5.1.1.

5.1.1 Platform Object Instantiation

In this section, we will specify the automatic mechanisms provided by the mediator to support platform object instantiation.

Platform objects, which include the more specific operating system and kernel objects, are logical objects provided by the platform. Platform objects include task, semaphores/mutexes, message queues, etc. These objects are usually created and used by the application dynamically. However, in a system like an AUTOSAR or an ARINC 653 platform, the objects cannot be created dynamically; instead, every application developer has to specify which and how many objects of a type the application requires. This information is then used by the integrator to statically create the objects needed during platform configuration.

Since the objects are only created during integration, the developer is unable to specify guarantees and configuration parameters for a

³¹ Logical in the sense that there is no counterpart in hardware

platform object directly. Instead, the platform developer specifies guarantee and configuration templates that describe the guarantees and configuration parameters a specific type of platform object must contain. The realization of this basic feature is provided by the mediator.

When the integrator specifies a new platform object, the mediator checks the service that provides the platform object for configuration and guarantee templates. If there are such templates, the mediator automatically creates a copy of every existing configuration parameter and every guarantee for the newly instantiated platform object. Now that the platform object contains its own set of parameters and guarantees, the mediator checks the guarantees for configuration-dependent conditions that still reference a template configuration parameter. If such a condition is found, the reference to the template parameter is replaced with a reference to the corresponding parameter of the platform object. After that, the platform integrator is able to configure the newly created platform object.

Figure 63 illustrates the automatic generation of platform object guarantees. The generation of configuration parameters is not shown, but is performed analogously to the guarantee generation.

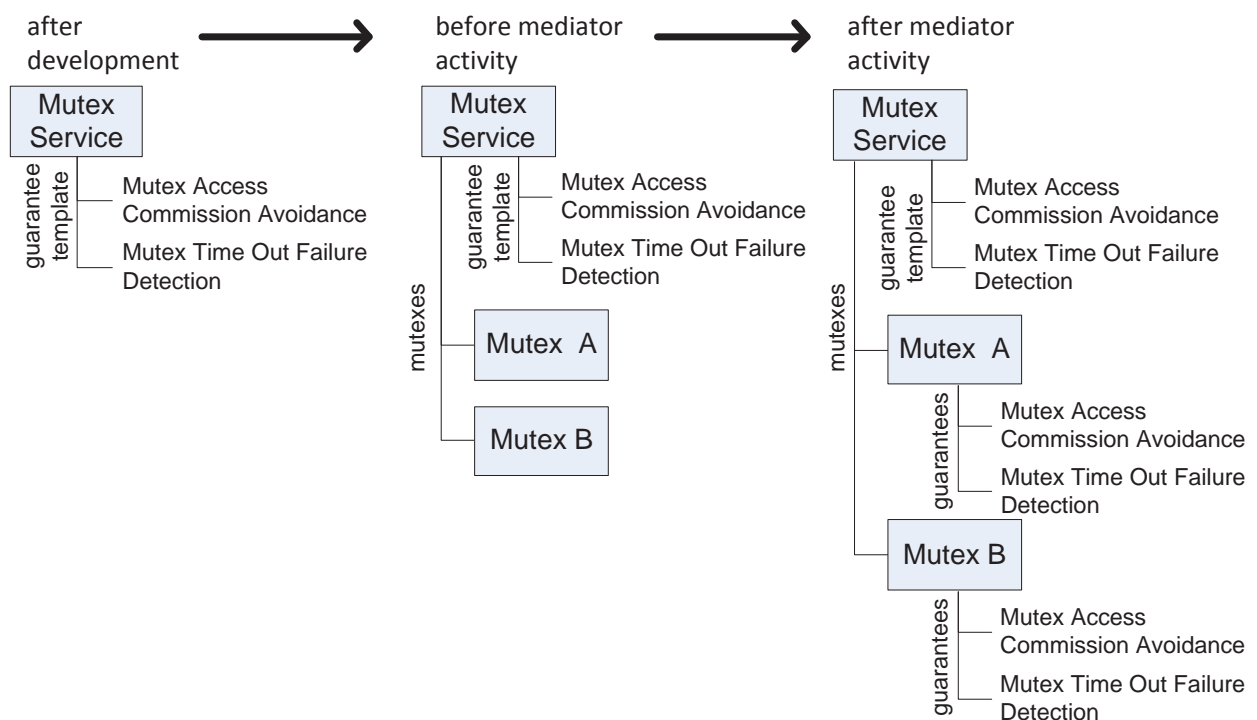


Figure 63: The process of automatic platform object guarantee generation

5.1.2 Evaluation of Configuration-Dependent Conditions

When the integrator finishes the configuration of an application or a platform, the application, respectively the platform, becomes ready for

integration. However, before the transition is triggered, the mediator checks the validity of the configuration; if the configuration is valid, the mediator automatically evaluates all configuration-dependent conditions specified in the corresponding safety interface.

The validity check basically validates that every configuration parameter that is used for the specification of at least one demand or guarantee is set to a valid value, which is a premise for evaluating configuration-dependent conditions. If the configuration is invalid, the mediator will not allow the transition to proceed to the integration phase.

However, if the configuration is valid, the transition is triggered and the mediator checks for the fulfillment of configuration-dependent conditions. The current version of the VerSal language supports three kinds of configuration-dependent conditions (see also section 4.3.2): an equals condition for integer parameters, an equals condition for Boolean parameters, and an equals condition for enumeration conditions. From our experience, most configuration parameters of a platform or application comply with one of the afore-mentioned types. However, especially regarding integer parameters, the implementation of greater than, less than, or “element of interval” conditions would be useful. Yet, we believe that the implementation of these conditions would have yielded no real research contribution, so we chose to omit this feature.

To check for fulfillment, the mediator checks for each configuration-dependent condition whether the required configuration value matches the actual configuration value. If it does, the status of the condition is set to “fulfilled”; otherwise the status of the condition is set to “violated”. However, the consequences of the condition evaluation are not analyzed during this analysis. Since the conditions affect mediation, this is done during interface mediation.

5.2 System Integration

The system integration step starts after the integrator has finished the configuration of applications and platforms. Now that the platform is appropriately configured to accommodate its guest applications, the integrator has to design the detailed mapping of the application’s resource needs to the resources provided by the platform. Thus, the reader should regard system integration as a refinement of the deployment plan. The deployment plan provides a coarse-grained mapping of application components to platforms and logical signals to communication links, whereas system integration specifies a much finer-grained mapping of, for example, memory sections (like `.text`) to memory regions or logical signals to bus messages.

Since the integrator's main task is to specify the afore-mentioned mappings, we provide a list showing every type of mapping in Annex A.4 (Table 16) in order to give the reader a feeling for the mapping's level of detail. The table specifies which application element (resource user) is mapped to which platform element (resource). For a detailed description of application and platform elements, we refer the reader to Appendix A.

When the integrator has specified this mapping, the VerSal mediator has all the information necessary to relate an application demand to the platform guarantees that can potentially fulfill this demand. This is possible because of the VerSal language's design. A demand is always attached to the application element that is directly affected by the platform's safety-related capabilities and a guarantee is always attached to the platform element that directly provides the safety-related guarantees. The mapping relates the application element to the corresponding platform element so that demands and guarantees can be related in a transitive manner as well. As an example, we look at an input signal provided by a sensor. The application's need for such a signal is specified by an input port. The application's demands regarding the failure behavior of the signal are attached to the demand as well. The deployment mapping, on the other hand, maps the input port to an input channel provided by the platform to model that the sensor signal is received via this particular channel. Finally, the platform's capabilities regarding the detection or avoidance of signal-related failure modes are attached to the input channel. Consequently, the VerSal mediator is capable of relating the demands to the relevant guarantees via the deployment mapping.

Before the mediator starts relating and evaluating demands and guarantees, however, it has to evaluate the deployment-dependent conditions, which will be described in section 5.2.1.

5.2.1 Evaluation of Deployment-Dependent Conditions

When the integrator finishes the system integration, the system becomes ready for mediation. However, before integration is finished, the mediator checks the validity of the mapping. Only if the integration is valid does the mediator evaluate the deployment-dependent conditions specified in the vertical interfaces of applications and platforms. A deployment mapping is valid if there is a mapping for every application element that must be mapped to a platform resource.

Regarding deployment-dependent conditions, the VerSal language offers dedicated extension points in the meta-model as well as in the implementation to efficiently allow extending the VerSal language. However, at the current point in time, the VerSal language does not

provide implementation for any deployment-dependent conditions. During our tests and during evaluation we did not need them, yet we wanted to offer the user of the language the possibility to use deployment-dependent conditions.

5.3 Manual Condition Evaluation

Conditions that cannot be evaluated automatically, like the generation of sufficient evidences, are modeled as manual conditions. Manual conditions have to be checked or directly fulfilled by the integrator.

A manual condition has to be sufficiently described by the creator of the conditions so that the integrator is capable of understanding its meaning. Additionally, the creator has to specify the required evidences for the condition fulfillment if there are any. During manual condition evaluation, the integrator then checks the description of the condition and, if the fulfillment depends on evidences, the integrator has to check the availability of the required evidences as well. In case the integrator is in charge of fulfilling the condition, the integrator generates the required evidences and, if possible, links the external document to the condition. In case the condition is to be fulfilled by a third party, the integrator can only check whether the condition has been successfully fulfilled or not. After fulfilling or checking the fulfillment of the condition, the integrator manually sets the status of the condition to “violated” or “fulfilled”. In the subsequent mediation, the automatic mediator handles manual conditions no different than other types of conditions.

5.4 Interface Mediation

The interface mediation step is triggered by the integrator after the evaluation of manual conditions. Together with the information provided during configuration and integration, the VerSal mediator is now capable of performing automatic demand mediation. The goal of the mediation is to check if every required application demand is fulfilled by the available platform guarantees and to provide information regarding the fulfillment, respectively violation, of demands. In this section, we will describe the process for checking demand fulfillment, while the information regarding the mediation result will be introduced in section 5.5, where we will describe how the integrator makes use of this information.

Mediation is performed separately for each application. Since an application can be spread over multiple platforms, the mediation of an application usually involves guarantees from several platforms. We want to note that even though each application is mediated separately, all the other applications in the system have to be configured and deployed

before mediation starts for any application. This is because other applications can influence the mediation of an application via shared platform resources³².

Overall, the mediation of an application contains three work steps: (1) demand assignment, (2) relevant guarantee retrieval, and (3) checking the fulfillment of the demand. Figure 64 gives an overview of the mediation process.

During demand assignment, the mediator retrieves an application demand from the list of application demands that have not been mediated yet, and identifies the type of the newly retrieved demand. Regarding demand-level mediation, we differentiate between the different demand classes known from chapter 4, which are platform service demands (section 4.5.1), health monitoring demands (section 4.5.2), resource protection demands (section 4.5.3), and service diversity demands (section 4.5.4). Since health monitoring demands are again divided into Application Monitoring Demands and Failure Reaction Demands, this leaves us with five different types of demands that have to be mediated in different ways.

In the second step, the mediator retrieves the guarantees relevant for fulfilling a particular demand. Relevant guarantees are mostly identified via the additional deployment information generated during system integration, but this process differs slightly from demand type to demand type and will therefore be described later. After the retrieval of the relevant guarantees, the mediator possesses all information to begin checking demand fulfillment.

During demand fulfillment, the mediator checks whether there is at least one guarantee in the set of relevant guarantees that is capable of fulfilling the demand at hand. This check for fulfillment depends strongly on the type of the demand as well, which is the main reason why we separated the mediation algorithm into five different sub-algorithms. However, all five mediation algorithms have in common that they are sub-divided into so-called prerequisite checks. Each prerequisite represents a certain characteristic that the guarantee has to fulfill in order to fulfill the overall demand, for example, integrity level sufficiency.

To conclude this introduction to interface mediation, we summarize that there are three different levels of mediation. Starting with the lowest level, there are prerequisites, demands, and applications. These levels interact as follows: A guarantee fulfills a demand if *every* demand-type-specific prerequisite is fulfilled. A demand, on the other hand, is fulfilled if there *exists one* guarantee that fulfills this particular demand. Finally,

³² Please note that only the architectural model of an application affects the mediation of other applications, not its demands.

the mediation of the application is successful if every required demand of the application is fulfilled.

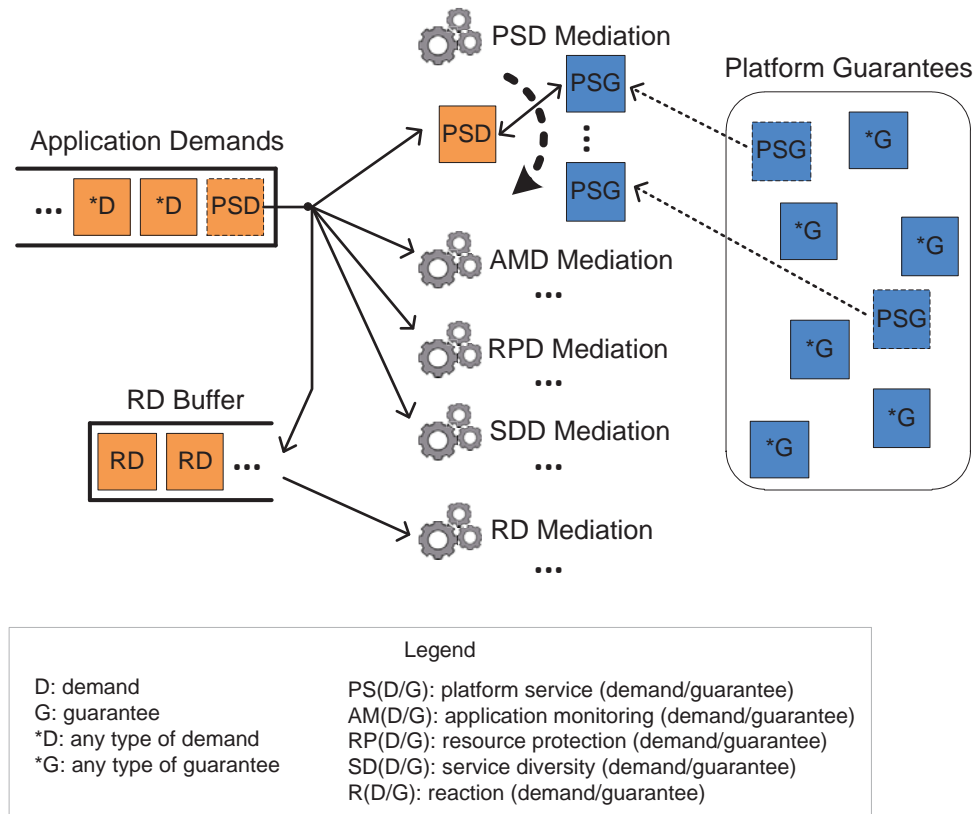


Figure 64: An overview of the automatic demand mediation process

In the following, we will describe the different mediation sub-algorithms that describe in more detail how demand fulfillment is decided. In section 5.4.1, we will describe the mediation of platform services. Sections 5.4.2 and 5.4.3 describe the mediation of application monitoring and reaction demands, both belonging to the health monitoring demand category. Section 5.4.4 contains the description of protection demand mediation, whereas section 5.4.5 describes resource diversity demand mediation. Section 5.4.6 finally describes various parameter-specific checks that are used in several instances.

5.4.1 Mediation of Platform Service Demands

In this section, we will describe the mediation of platform service demands, i.e., the automatic process provided by the VerSal mediator that checks whether a platform service demand is fulfilled or violated by the available platform guarantees.

To recapitulate, a platform service failure demand enables the application developer to specify demands regarding the avoidance or detection of platform service failures. The application developer can, for

example, demand that the corruption of a certain logical signal is detected or avoided. The mediation of a platform service demand is successful if the corresponding platform service, e.g., the communication link transporting the signals, provides a sufficient guarantee. Checking for the availability of a sufficient guarantee involves checking ten prerequisites, which are described in the following. An overview of platform service demand mediation is provided in Figure 65.

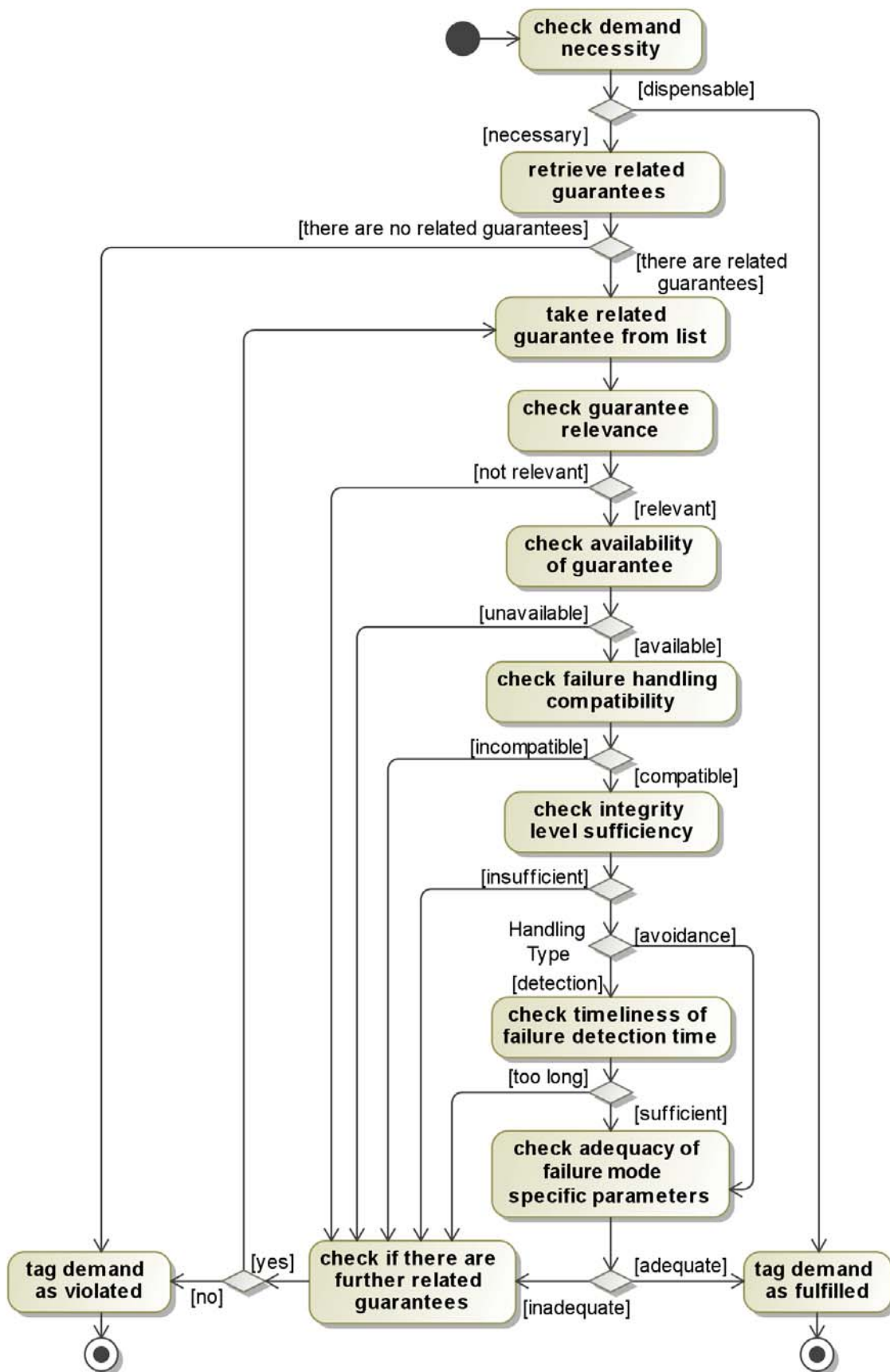


Figure 65: An overview of platform service demand mediation

The first prerequisite that has to be checked when mediating a demand is whether the demand is actually necessary. Since applications are configurable, a demand might not be required in every possible configuration of the application, which is why the VerSal language allows the application developer to specify conditional demands. The conditions are directly attached to the demand and only if the conditions evaluate to true is the demand necessary. If the conditions evaluate to false, the demand is regarded as dispensable and does not have to be mediated. This mediation semantics is represented by directly setting a dispensable demand to “fulfilled” so that it does not interfere with the mediation of the residual demands. Specifying a conditional demand is optional, and if there are no conditions attached to a demand, the demand is regarded as necessary and its fulfillment has to be checked.

The next step in mediating a platform service demand is the identification of related guarantees. In brief, a guarantee is related to a demand if the guarantee is provided by the service that the owner of the demand is deployed to. As an example let us regard an input failure demand. The input failure demand is owned by/contained in the potentially faulty input signal. This signal is deployed to an input channel during system integration, and this input channel provides guarantees. These guarantees are called the related guarantees of our example demand. Figure 66 visualizes this transitive relation dependency. In order to retrieve related guarantees, the mediator checks the deployment information of the demand owner. If the corresponding application element is not deployed correctly, the demand cannot be mediated and is considered as violated. The same is true if the resulting platform element contains no guarantees. However, if there are related guarantees, each of these guarantees is checked individually.

The first prerequisite that is checked for a guarantee is its relevance. A related guarantee is labeled relevant if it possesses some basic characteristics required for fulfilling the demand at hand. Since a platform element can contain different kinds of guarantees, the mediator first checks whether the type of the guarantee matches the type of the demand, i.e., if the guarantee is also a platform service guarantee and not, for example, a resource protection guarantee. In the second step, the mediator checks if the failure mode of the guarantee matches the failure mode of the demand. If the failure modes match, the mediator knows that the platform element is generally capable of handling the demanded failure mode. In the next steps, the mediator has to check whether the guarantee allows handling the failure in a sufficient way.

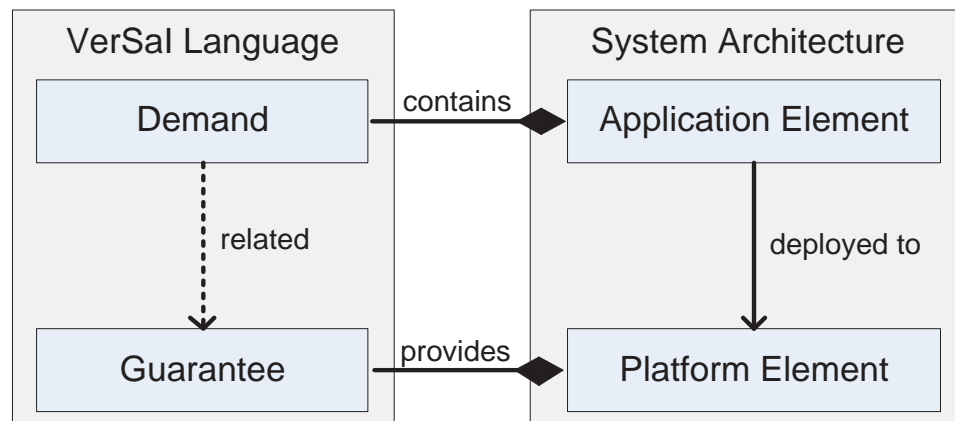


Figure 66: The concept of related guarantees

Prior to assessing the specific guarantee capabilities, the mediator checks whether the guarantee is available. Comparable to the conditional necessity of a demand, certain platform guarantees have a conditional availability, usually depending on the configuration parameters. If the guarantee is conditional and the conditions are not fulfilled, the guarantee is not available in the current configuration and therefore incapable of fulfilling the demand. If the guarantee is available, the mediator continues the mediation by checking the compatibility of the guarantee's failure handling type.

As described in section 4.5.1, the VerSal language differentiates between failure detection and failure avoidance. A failure detection guarantee specifies the platform's capabilities of detecting a failure, which provides the basis for a certain reaction, e.g., a failure indication. On the other hand, a failure avoidance guarantee specifies the capability of completely avoiding a certain failure mode, for example, due to the design of the corresponding platform service or the availability of internal failure correction mechanisms. If the application demands detection of a failure mode, this can be fulfilled by a detection guarantee or by an avoidance guarantee, since the absence of a certain failure renders the demand for its detection void. Conversely, an avoidance demand can only be fulfilled by an avoidance guarantee. If the failure handling type matches, i.e., if an avoidance demand is fulfilled by an avoidance guarantee, the mediation continues with a check for integrity level sufficiency.

Integrity level sufficiency requires that the integrity level of the guarantee is at least as high as the integrity level of the demand. The integrity level of a guarantee is a reflection of its trustworthiness and sometimes also of certain capabilities of the guarantee. Regarding trustworthiness, the integrity level defines the rigor with which the guarantee has been implemented and verified. The higher the integrity level of the guarantee, the higher the trustworthiness of its correct implementation. In addition to trustworthiness, certain standards loosely attach

capabilities to the integrity level as well, for example the likelihood with which a failure is detected or avoided. Only if the guarantee at least equally matches the demand's integrity level is the guarantee capable of adequately handling the demand.

The next prerequisite to be checked is the timeliness of the detection mechanism promised by a detection guarantee. This check is, of course, only relevant if the demand is a detection demand and if the current guarantee is a detection guarantee. If a detection demand is to be fulfilled by an avoidance demand, there will be no detection and consequently, no detection time. In case this check is required, the detection time specifies the maximum time between failure occurrence and failure detection. The mediator has to assess whether the promised detection time is lower than the required detection time. If the guarantee passes this prerequisite check, the mediator has to check the adequacy of the failure-mode-specific parameters.

Most failure modes of the VerSal language have additional parameters to allow the user to specify failure modes on a more detailed level. In case the failure mode is used in the context of a demand, these parameters allow the application developer to precisely specify the border line between acceptable and erroneous behavior. In case the failure mode is used in the context of a guarantee, these parameters allow the platform developer to precisely specify the capabilities of the platform regarding the detection or avoidance of the failure mode. Example parameters are jitters, error levels (as a measure for the deviation from the correct behavior), or delays. Since parameters are failure-mode-dependent and different types of parameters are mediated differently, we will describe parameter-specific mediation in a separate section (see 5.4.6).

If the failure-mode-specific parameters of the guarantee are sufficient for fulfilling the failure-mode-specific parameters of the demand, the guarantee is finally assessed as being capable of fulfilling the demand. The demand is tagged as fulfilled and the mediation algorithm continues with the mediation of the next demand.

In order to exemplify the mediation of platform service demands, we will use the example demand D2, which was defined in section 4.5.1. The demand is contained in the output port **a_set_fin** of our running example application and reads as follows:

*Example D2: A value failure of the output signal **a_set_fin** larger than 0.05V must be detected within 0.05ms (ASIL C).*

As specified in section 4.1, **a_set_fin** is deployed to the platform output channel **voltage_out**. Let us assume that this output port

provides three different guarantees, a resource protection guarantee and two platform service guarantees to address latency and value failures.

In the first step of the mediation, the mediator checks the necessity of D2, but since D2 is unconditional, it is necessary per default. The second step is the retrieval of related guarantees. To do so, the mediator queries the architectural model of our example to retrieve the deployment information of **a_set_fin**. In our case, the query returns with the information that **a_set_fin** is deployed to the output channel **voltage_out**. A subsequent query provides the mediator with the three guarantees that are provided by **voltage_out**, which are individually analyzed by the mediator.

If we assume that the mediator first selects the resource protection guarantee for evaluation, the check for guarantee relevance fails, since a platform service demand like D2 can never be fulfilled by a resource protection guarantee. The second guarantee that the mediator checks fails as well. The guarantee is a platform service guarantee, but the failure modes do not match (output latency vs. value failures). However, the final guarantee contained in **voltage_out** is a platform service guarantee that addresses value failures and that reads as follows:

Example G10: A value failure of an output signal issued via voltage_out larger than 0.02V is detected within 0.03ms (ASIL C). Conditions apply: "The configuration parameter enable_detection must be set to true".

Since G10 is a platform service guarantee that addresses value failures, it passes the relevance test. Following the relevance test, the mediator checks the availability of G10. Since G10 is a conditional guarantee, it is only available if the corresponding configuration parameter **enable_detection** is set accordingly. Since we assume that the configuration parameter is set to true, the condition is evaluated as "fulfilled" and consequently, the guarantee is available.

The mediation continues with its sixth step, the failure handling capability check, which is successful since D2 demands failure detection and G10 provides failure detection. The integrity level check is also successful as G10 is provided with **ASIL C** integrity and D2 demands **ASIL C** integrity. Since G10 is a detection guarantee (as opposed to an avoidance guarantee), the appropriateness of the detection time has to be checked as well. As the provided time interval (**0.03ms**) is shorter than the demanded time interval (**0.05ms**), the timeliness check is successful and the mediation reaches its last stage, in which the failure-mode-specific parameters are evaluated. In our example, the mediator has to check the allowed deviation parameter of the analog value failure mode. The tolerated deviation as specified by D2 is **0.05V**, whereas the maximum deviation provided by G10 is **0.02V**. As a consequence, the

mediator decides that the example demand D2 is adequately fulfilled by G10.

5.4.2 Mediation of Application Monitoring Demands

In this section, we will describe the mediation of application monitoring demands, i.e., the automatic process provided by the VerSal mediator that checks whether an application monitoring demand is fulfilled or violated by the available platform guarantees.

To recapitulate, an application monitoring demand enables the application developer to strengthen the fault tolerance of the application at hand. The application monitoring demand requests the platform to detect a deviation from the application's nominal behavior, i.e., an application failure. The platform has the capability of monitoring the behavior of the application since the platform is involved in the realization of most of the application's functionality. The platform typically provides general-purpose, configurable monitoring mechanisms. These are adapted and configured by the integrator to detect the failures of guest applications. The algorithm that checks for the availability of a guarantee that is sufficient for fulfilling an application monitoring demand involves checking eight prerequisites. This process will be described in the following; an overview of the platform service demand mediation is provided in Figure 67.

The mediation of an application monitoring demand is in many ways comparable to the mediation of a platform service demand. More precisely, it is comparable to the mediation of a platform service *detection* demand, since there are no application monitoring avoidance demands. This comparability is reflected by the fact that seven of the eight prerequisite checks involved in application monitoring mediation are also used for platform service demand mediation. Instead of describing these checks redundantly, we will refer to the previous section where applicable.

Application monitoring demand mediation starts with checking the necessity of the demand at hand. Only if the demand is necessary does it have to be mediated; otherwise it is automatically tagged as "fulfilled" and taken out of the mediation. The mediator then identifies the related guarantees via the deployment mapping of the demand owner and begins the iterative assessment of the individual related guarantees. The first step of guarantee-centric mediation is to verify that the related guarantee is also relevant. Since application monitoring demands are sub-typed via application failure modes (see section 4.4.3), the relevance check involves checking if the guarantee is an application monitoring guarantee and if the guarantee's failure mode matches with the failure mode provided by the demand. If that is the case, the guarantee is

checked regarding its availability and the sufficiency of its integrity level. These five prerequisite checks are performed during platform service demand mediation as well, and a more detailed description of the checks is found in section 5.4.1, where platform service demand mediation is introduced.

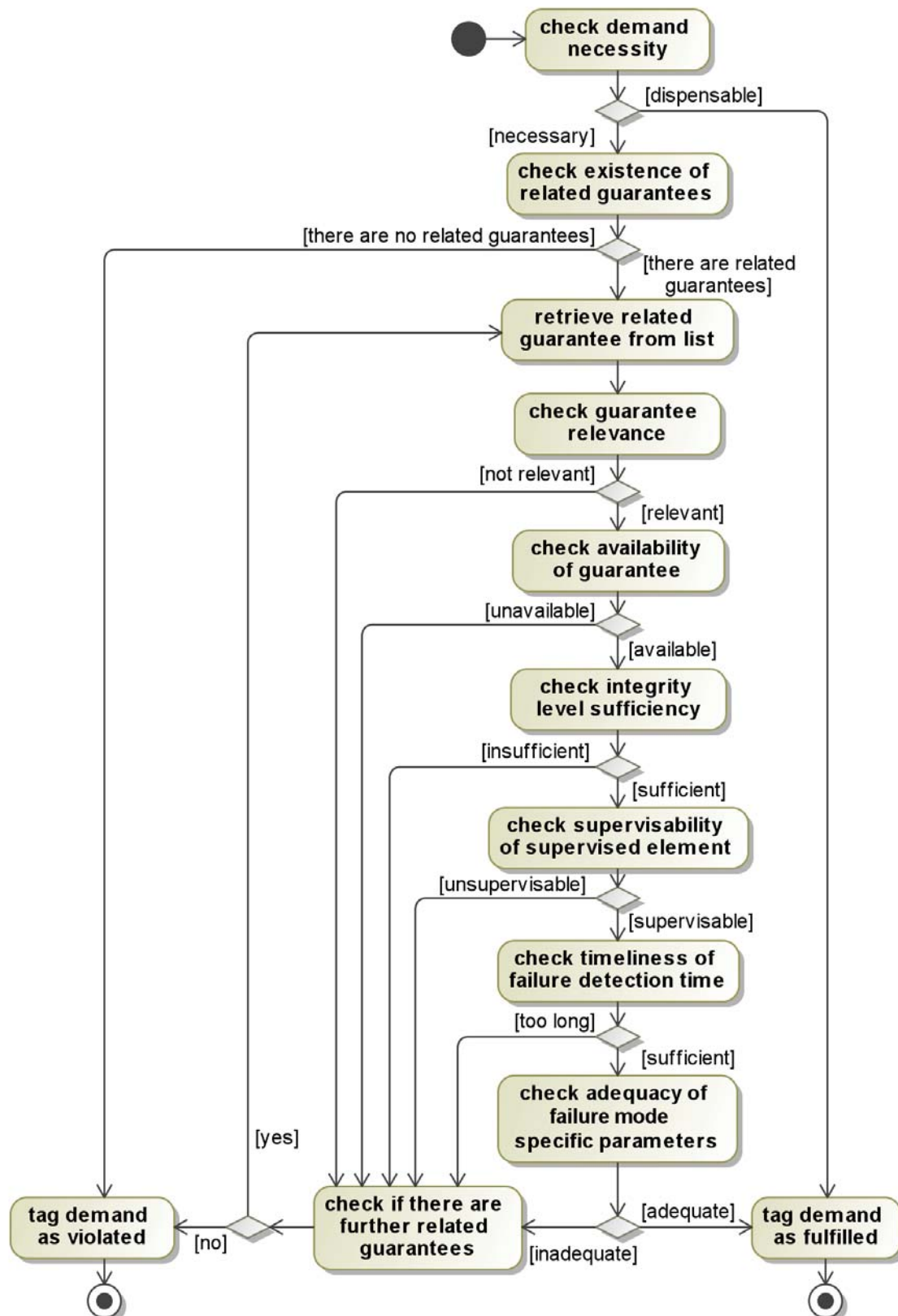


Figure 67: An overview of the application monitoring demand mediation

The check following thereafter is unique to application monitoring demands. An application monitoring demand is not owned by the

monitored/supervised entity but by a so-called application monitoring need, which is deployed to an application monitoring service that provides the corresponding monitoring guarantees. Therefore, the supervised entity is identified via a reference from the guarantee. Every executable entity can be supervised, which includes time-triggered runnables, event-triggered runnables, and ISRs. However, the platform is not necessarily capable of detecting the corresponding failure mode for every executable. Therefore, the mediator has to check whether the platform is capable of supervising the relevant executable before mediation continues. If the required executable entity cannot be supervised by the platform's monitoring service, the guarantee is unable to fulfill the demand.

In case the executable entity is supervisable, the mediator has to check the sufficiency of the failure detection time specified by the guarantee. Since the platform cannot guarantee the avoidance of application failures, there are no avoidance demands for this demand type. Consequently, the detection time check is mandatory for all demands of this type. Since application failures have parameters as well (comparable to platform service failures), the last step in application monitoring demand mediation is to check the adequacy of the failure-mode-specific parameters, which is described in a separate section of this chapter (see 5.4.6).

If the parameters of the guarantee are adequate, the demand is tagged as fulfilled and the mediation continues with another demand.

The mediation of application monitoring demands is comparable to the mediation of platform service demands, which is also shown by the following example. The following example demand is specified by the software component `v_controller` known from our running example. The component contains a specific port for the specification of so-called service needs, which again contains the example demand D4 already specified in section 4.5.2.

Example D4: The platform must detect an execution time of the executable `v_controller_main` of more than 0.016ms (ASIL C).

A service need specified by an application is always deployed to the corresponding service provided by the platform. In this case, the monitoring service need is deployed to the **health monitoring service** provided by our example platform. With the help of this deployment information, the mediator is capable of retrieving the related guarantees contained in the corresponding service, one of which is example guarantee G3 known from section 4.6.2.

Example G3: The platform is capable of detecting execution time failures for runnables (ASIL C).

From this point on out, the mediation of D4 is comparable to the mediation of platform service demands. The mediator checks whether the failure types match (in this, both failure types are “execution time failures”), whether the guarantee is available (there is no condition and therefore G3 is available), whether the integrity level is the same (both ASIL C), whether timeliness is guaranteed (not relevant here), and whether the failure-mode-specific parameters are fulfilled (the parameter specified by D4 is fulfilled since G3 does not restrict the precision of its monitoring facility).

However, in addition to these items, the supervisability of the supervised element, in this case the executable `v_controller_main`, has to be checked. In our case the platform is only capable of supervising the execution of runnables, but not the execution of ISRs. Consequently, the mediator has to check whether `v_controller_main` is of the type runnable (which it is) before knowing whether D4 can be fulfilled by G3.

5.4.3 Mediation of Failure Reaction Demands

In this section, we will describe the mediation of failure reaction demands, i.e., the automatic process provided by the VerSal mediator that checks whether a failure reaction demand is fulfilled or violated by the available platform guarantees.

To refresh the basic semantics of a failure reaction demand, the application developer specifies a failure reaction demand to request a failure control reaction from the platform. In an integrated architecture, the application is only allowed to use the platform API to interact with the platform software and hardware. This design restriction protects the system from erroneous applications but also limits the application’s freedom to perform certain failure recovery reactions. Therefore the application uses the platform to explicitly trigger these restricted recovery reactions. In addition to this motivation for using failure reaction demands, the platform can be seen as an element independent of the application. When an application-related failure occurs, this may render the affected application unable to perform a reaction, whereas the platform might still be able to react. The algorithm for mediating platform failure reactions is the most complicated of all five demand classes and contains ten prerequisite checks. The algorithm is described in the following and is depicted in Figure 68.

The mediation of a failure reaction demand starts with the regular check for the demand’s necessity. If the demand is necessary, the subsequent steps depend on how the reaction demand is triggered. As described in

section 4.5.2, there are request-triggered reaction demands and detection-triggered reaction demands. A request-triggered reaction is triggered directly by the application by calling the respective API function. Conversely, a detection-triggered reaction is automatically triggered by the platform if a specific failure is detected. In the latter case, the detection that triggers the reaction is identified via a reference from the reaction demand to the corresponding detection demand. Since the correct execution of the reaction depends on the correct detection of the failure, the reaction demand can only be mediated if the trigger demand has been successfully mediated as well. To be able to check that, reaction demand mediation is delayed until potential detection demands are mediated (see Figure 64).

If the corresponding trigger demand is successfully mediated, an additional aspect related to the trigger demand has to be checked. If the trigger demand is a platform service demand, we have to check whether the trigger demand has been fulfilled by an avoidance guarantee. In this case, the failure that is supposed to trigger the reaction will never occur, which renders the detection-triggered reaction demand useless. Such a reaction demand is regarded as dispensable, comparable to an unnecessary conditional demand, and is marked as fulfilled by default.

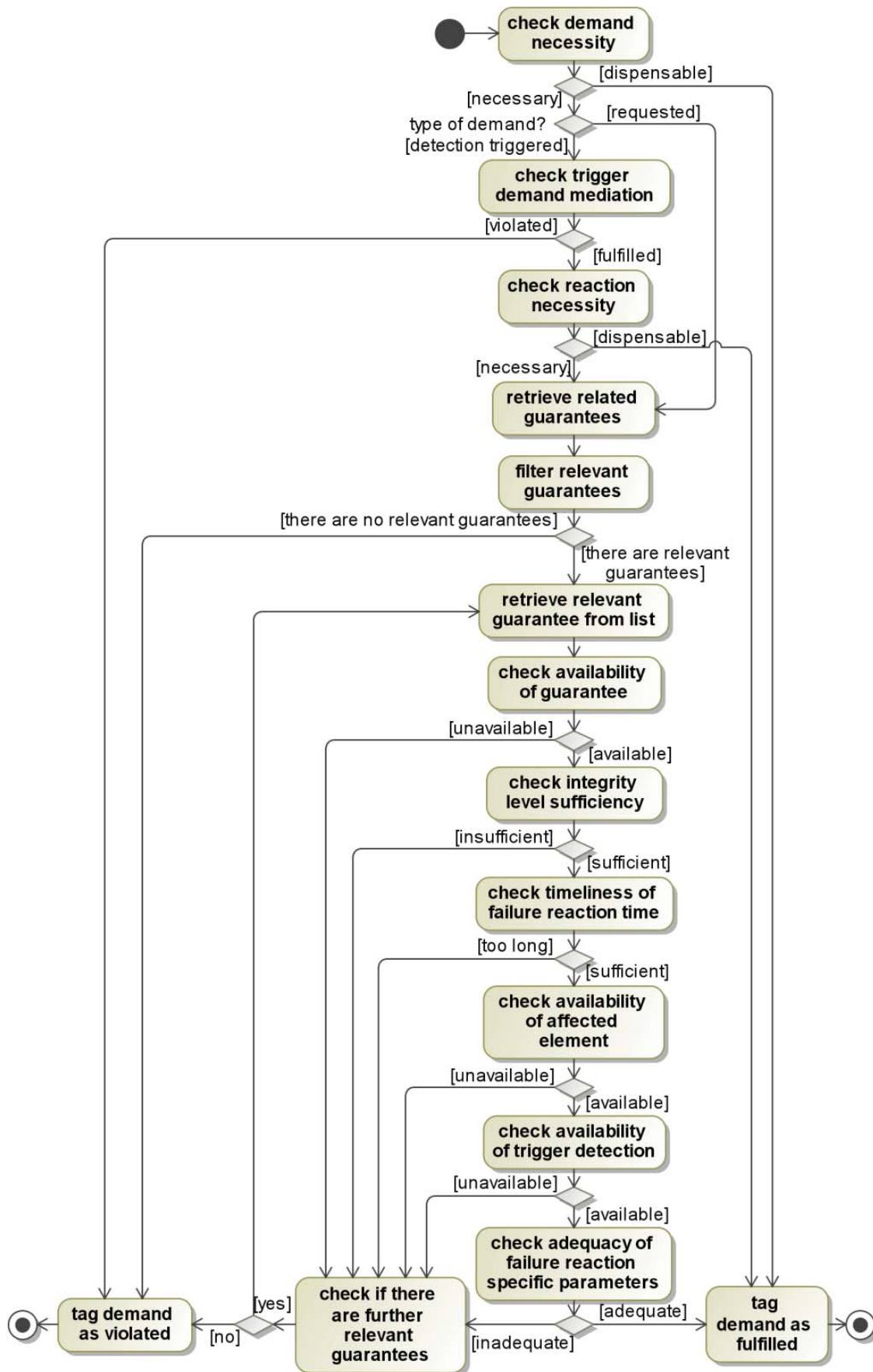


Figure 68: An overview of failure reaction demand mediation

The next steps have to be performed equally for both request-triggered and detection-triggered demands. First, the mediator retrieves the related guarantees via the deployment of the demand's failure reaction need to one of the platform's failure reaction services. Filtering for relevant guarantees is performed via the demand's platform failure reaction type (see section 4.4.4), which has to match with the type of the guarantee's failure reaction. Following this filtering, the guarantees are, as usual, checked individually regarding their ability to fulfill the demand at hand.

At first, each guarantee is checked regarding its availability and its integrity level sufficiency; both checks are described in more detail in section 5.4.1. After that, the timeliness of the failure reaction time is checked. The failure reaction time is the time between the detection of the failure and the end of the failure reaction. The failure reaction time and the failure detection time are usually chosen so that the sum of both is smaller than the tolerance time of the failure. The failure reaction time is regarded as sufficient if the failure reaction time assured by the guarantee is smaller than the failure reaction time required by the demand.

Every failure reaction specifies a so-called affected element. This platform element is directly affected by the reaction, like a partition or a task that is shut down as a result of the reaction, or an output channel that is set to a fail-safe state. The affected platform element required by the demand is identified via the demand's affected application element. If, for example, the demand specifies that a certain runnable or ASWC has to be restarted, or that a logical signal has to be set to its default value, this is translated into a restart of the task that hosts the runnable, a restart of the partition that hosts the ASWC, or into setting the output channel that emits the logical signal to its default value, respectively.

Since the platform developer may design the platform such that not every element is capable of performing the required reaction, the mediator has to check whether the required platform element is able to perform the demanded reaction. This information is specified by the guarantee since the failure reaction guarantee references every platform element that is capable of performing the relevant reaction. Only if the required platform element is in the list of the available platform elements can the guarantee execute the demanded reaction and fulfill the demand at hand.

Comparable to design restrictions regarding the platform elements available for implementing reactions, there are certain platforms where not every kind of failure is capable of triggering every kind of reaction. In an ARINC 653 compatible platform, for example, failures are categorized according to their potential influence and only the failures that are able to harm the whole platform are allowed to trigger a restart of the

platform. To model this behavior, a detection-triggered failure reaction guarantee references all failures that are capable of triggering the reaction. If the demand at hand is a detection-triggered failure reaction demand, the mediator checks whether the failure specified by the demand is capable of triggering the required guarantee, and only if this is possible can the guarantee fulfill the demand.

If all the previous prerequisite checks have been successfully passed, the mediator finally checks for the adequacy of the guarantee's reaction-specific parameters. If these parameters are suitable for fulfilling the reaction-specific parameters of the demand, the guarantee is finally marked as being able to fulfill the demand. Comparable to the parameters of platform service failures and application failures, the mediation of parameters specific for different types of reactions is described in section 5.4.6.

To exemplify the mediation of failure reaction demands, we discuss the mediation of example demand D12.

Example D12: Upon detection of the output value failure of signal `a_set_fin`, the platform must shut down the partition that hosts the software component `throttleSWC` within 0.05ms (ASIL C).

The requirement is contained in a service need of the software component `throttleSWC` and demands the shutdown of its own partition in case a value failure of the output signal `a_set_fin` is detected. The first step in the mediation of this demand is to check whether there is a corresponding trigger demand that requests the detection of the value failure and, if such a demand exists, whether that demand has been successfully fulfilled. As described in section 5.4.1, there is such a demand, namely D2, and the demand has been successfully mediated.

In the next step, the related guarantees are identified via the deployment of `throttleSWC`'s service need port to the **health monitoring service** of the example platform. This process yields the related and relevant guarantee G6.

Example G6: The platform is capable of shutting down partitions upon detection. Possible triggers: all failure detection events. Possible restart targets: partition objects with the configuration condition "`restart_enabled == true`". (ASIL C) Conditions apply: "The caller must have sufficient rights to request the shutdown".

The availability of the guarantee depends on a condition that has to be manually checked by the integrator since the VerSal method is not capable of checking rights and permissions in its current version. Let us

assume that the integrator marks this manual condition as fulfilled, so that the mediator is capable of continuing with the subsequent checks.

The following checks for integrity level sufficiency and for timeliness of the failure reaction time (comparable to the check for timeliness of failure detection time) were already covered by previous examples and hold no special points of interest. In the next step, however, the mediator has to check for the availability of the affected element. In our case, the affected element is **partition 2**, since **throttleSWC** is deployed to this partition. In order to find out whether partition 2 can be shut down, the mediator has to evaluate whether the configuration parameter **restart_enabled** is set to **true**.

If we assume that this is the case, the mediation continues by checking whether an “output value failure”, as demanded by D12, is capable of triggering a partition shutdown in our example platform. Since G6 specifies no restrictions regarding the possible trigger events of partition shutdowns, D12 is fulfilled by our example platform.

5.4.4 Mediation of Resource Protection Demands

In this section, we will describe the mediation of resource protection demands, i.e., the automatic process provided by the VerSal mediator that checks whether a resource protection demand is fulfilled or violated by the available platform guarantees.

Resource protection demands are modeled by the application developer to demand protection from so-called interferences caused by the sharing of platform resources among mixed-critical applications. An interference is a special type of failure scenario, which is characterized by the following cause-effect chain: At the beginning of an interference, an application uses a shared platform resource, typically in an erroneous manner (e.g., it uses it for too long or modifies it in the wrong way). This resource utilization affects the resource in such a way that it is unable to provide its service as demanded by another application. This other application is affected by the misbehavior of the causative application, as it perceives a failure of the affected platform resource. These failures can be comparable to those specified in subsection 4.5.1. Via this additional failure propagation channel, applications can interfere with each other even if there is no functional dependency between the corresponding applications. To fulfill a resource protection demand, the platform has to adequately protect shared resources from interferences. Whether the provided protection mechanisms are adequate is checked by the mediator by performing nine individual checks. An overview of the resource protection demand mediation is provided in Figure 69.

The mediation of a resource protection demand starts as usual, with a check regarding the necessity of the demand. After that, the mediator checks for the necessity of protection. Protection is only necessary if the current application shares the platform with other applications that have lower criticality than the application's own criticality. Consequently, if there are no other applications with lower criticality, protection is not necessary and the mediator tags the demand as fulfilled by default.

If protection is necessary, the mediator assesses whether there are related guarantees. Comparable to platform failure demands, protection demands are contained in the application element that is affected by the potential interference. Communication interference demands are, for example, contained in communication ports just like communication failure demands. Analogously, resource protection guarantees are contained in the platform resource that is potentially affected by the interference as well and that propagates the interference effect to the application element using the resource. To stay with the previous example, a communication protection guarantee is contained in a communication link. Since every protection guarantee contained in the related platform element is relevant for the mediation of the demand, we do not differentiate between relevant and related demands as we did for the mediation of the previous three demand types.

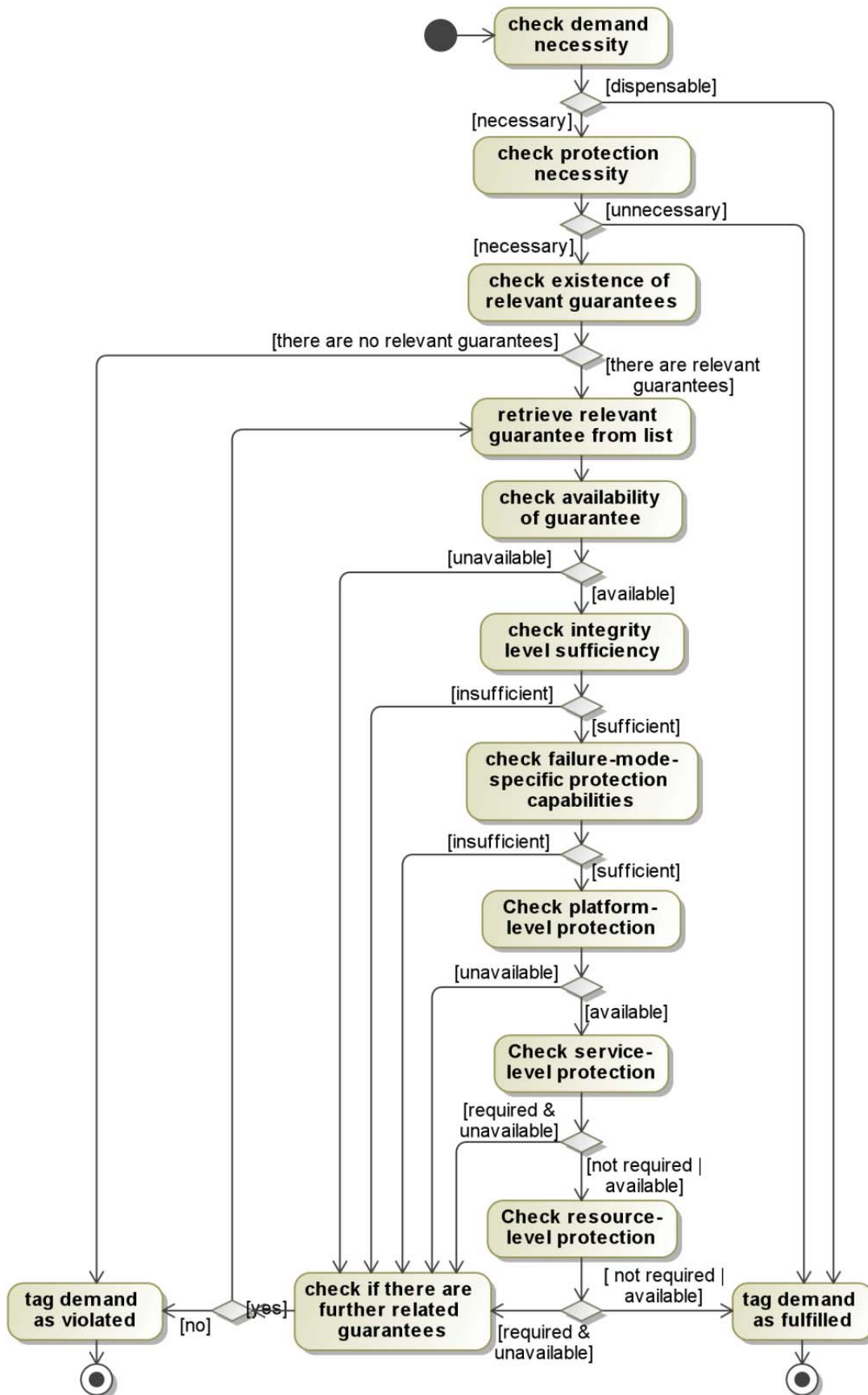


Figure 69: An overview of resource protection demand mediation

Subsequently, each relevant guarantee is, as usual, checked for its integrity level sufficiency. If the integrity level is sufficient, the mediator checks whether the platform is capable of protecting against every relevant failure mode. If the application developer demands protection, it is possible that protection is not required regarding every possible failure. It is, for example, possible that a value failure is safety relevant, but a late failure is not. Consequently, the platform does not have to provide protection regarding every possible failure mode either to fulfill a protection demand. To model the critical failure modes against which protection is necessary, the application developer specifies a list of related failure modes that are then contained in the demand. Analogously, the platform developer specifies a list of failure modes that are included in the protection. To do so, both developers choose from the failure modes provided by the VerSal language (see section 4.4.2). If the specified set of demand failure modes is a subset of the guarantee's failure modes, the check is successful.

The last three checks of resource protection mediation assess the sufficiency of the given protection on three levels: platform level, service level, and resource level.

On the platform level, the check evaluates if the resource is automatically protected from an ASWC with lower criticality or whether lower-criticality ASWCs have to be allocated to a different partition than the ASWC that owns the demand. From the previously performed protection necessity check we know that there is at least one less critical ASWC deployed to the platform. If the platform guarantee currently being checked specifies that the resource can only be protected from ASWCs that belong to another partition, we have to check if the partition of the demand owner ASWC does not contain any less critical ASWCs. If partitioning is not required or if there are no lower-criticality ASWCs in the relevant partition, the mediator continues with the evaluation of the protection on the service level.

On the previous level we checked for the protection of potentially unrelated ASWCs, which are those ASWCs that neither directly use the relevant resource nor the service that provides the resource. On the service level, we check whether the platform is capable of protecting the resource from interferences if there are less critical ASWCs that use the same service that provides the resource. Please note that only those resources that we label as platform object (mutexes, timers, tasks ...) are actually provided by services. Resources like cores or input channels are directly provided by the platform, and service-level protection is not applicable to those resources. If the platform is capable of providing protection against lower-criticality ASWCs that share the service or if there are no such lower-criticality ASWCs, the mediator continues with the final resource-level protection check.

During this last step, the mediator checks whether there are less critical ASWCs that share the resource with the demand owner ASWC, and whether the platform is capable of protecting the related resource from interference by these ASWCs. If protection is available or if it is not required (if there are no lower-criticality ASWCs that share the resource), the guarantee is finally assessed as being capable of fulfilling the demand at hand.

In the following, we will exemplify the mediation of resource protection demand using the example demand D10 previously specified in section 4.5.1. D10 is contained in the **monitoring** ASWC, more precisely in service need **sn6**, and reads as follows:

Example D10: The monitoring.error_event service need must be protected from interferences that cause the failure modes Event Signal Commission, Event Timeout Failure (ASIL C).

Unlike the previously described mediations, the mediation of resource protection demands depends heavily on the deployment of other application elements. Accordingly, in the first step of the mediation the mediator checks whether there are less critical components deployed to the same platform. In our case, there are such components, namely **GUI**, **v_sensorSWC_A**, and **v_sensorSWC_B**. Consequently, protection is principally necessary and D10 is not fulfilled by default. In the following step, the mediator retrieves the relevant guarantees as usual via the deployment of **sn6** to the **event service** of our example platform, which yields the following guarantee previously specified in section 4.6.3:

Example G8: The platform is capable of protecting the service event service from interferences that cause the failure modes Event Signal Commission, Event Signal Omission, Event Timeout Failure. Mixed-critical users do not have to be allocated to different partitions. Mixed-critical users are allowed to use the same service. Mixed-critical users are not allowed to use the same event (ASIL C).

G8 is available since it is unconditional; it also has a sufficient integrity level (ASIL C vs. ASIL C), which is why the guarantee passes the next two standard mediation checks. In the following step, the mediator checks whether the platform protects the event service from all relevant failure modes. In our case, the protected failure modes are a true subset of the failure modes that need to be protected and consequently, the guarantee passes this test as well.

In the next three stages, the mediator has to inspect the deployment of components more closely to decide whether G8 fulfills D10. On the platform level, the mediator checks whether the provided protection

requires partition separation and if it does, whether the relevant partition contains mixed-critical components. In the current case, however, the protection neither requires partition separation nor does **monitoring's** partition contain mixed-critical components. As a consequence, G8 passes the check on the platform level. On the service level, the mediator checks whether the service can be protected from mixed-critical users. According to G8, the event service can be protected from mixed-critical users, which is also necessary since the QM-rated **GUI** ASWC uses the event service as well. Finally, the mediator checks whether protection can be guaranteed on the event level. For the example situation, the mediator finds that the corresponding event called **error_event** is used by both GUI and monitoring, but according to G8, element-level protection is not guaranteed. Consequently, G8 is not capable of fulfilling D10.

5.4.5 Mediation of Service Diversity Demands

In this section, we will describe the mediation of service diversity demands, i.e., the automatic process provided by the VerSal mediator that checks whether a service diversity demand is fulfilled or violated by the available platform guarantees.

The application developer specifies a service diversity demand to support a redundant two-channel architecture of the application. With the specification of a diversity demand, the developer demands that two specific channels (input, output, or communication) used by the application are developed diversely by the platform. Such a demand is fulfilled if the platform is capable of providing the relevant channels diversely, i.e., if the relevant channels fail independently with regard to systematic failures. The corresponding mediation algorithm contains six checks and is shown in Figure 70.

After the regular check for the necessity of the demand, the mediator retrieves the demand's related guarantees. However, the retrieval of related guarantees is not as straightforward as it was for the previous demand types. Service diversity demands are contained in applications and the corresponding guarantees are contained in platforms. Since an application is not directly deployed to a platform (the software components of the application are), we cannot simply follow the deployment of the demand owner to retrieve the related guarantees. Instead, the demand references the application elements (e.g., communication ports) that have to be deployed to diverse channels (e.g., communication links). The mediator tries to retrieve the related guarantees via the deployment of these application elements. Yet, it is possible that both application elements are deployed to resources provided by different platforms. Since there are no guarantees that span different platforms, there are no matching diversity guarantees and the

mediator cannot mediate the demand with the given deployment. In this case, the mediator tags the demand as fulfilled under the premise that a manual check shows the diversity of the redundant channels. In case a single relevant platform can be identified, the related guarantees can be retrieved.

In the next step, the mediator filters the relevant guarantees from the related guarantees. Since there are three types of diversity demands (input channel diversity, output channel diversity, and communication channel diversity), the mediator assesses only those diversity guarantees that match the type of the diversity demand.

After the usual check for the guarantee's availability, the mediator checks whether the channels provided by the guarantee match the channels required by the demand. The required channels are identified via the deployment of the application elements referenced by the demand. If these required channels are identical to the channels referenced by the guarantee, the relevant channels were indeed developed diversely and the mediation continues.

Comparable to resource protection demands, service diversity demands are also specified in a failure-mode-specific manner. When two platform resources are developed diversely, it is also possible that the independence of a certain failure mode cannot be guaranteed. Analogously, it is possible that the application developer does not demand independence with regard to every possible failure mode of the related resource. Therefore, resource diversity demands as well as resource diversity guarantees reference a list of failure modes, namely those failure modes that are required to occur independently and those failure modes that are guaranteed to occur independently. Only if the set of required independent failure modes is a subset of the guaranteed independent failure modes does the mediation of the guarantee continue.

The last check the guarantee has to pass is the check for its integrity level sufficiency. If the integrity level is sufficient, the guarantee is finally assessed as being capable of fulfilling the demand at hand.

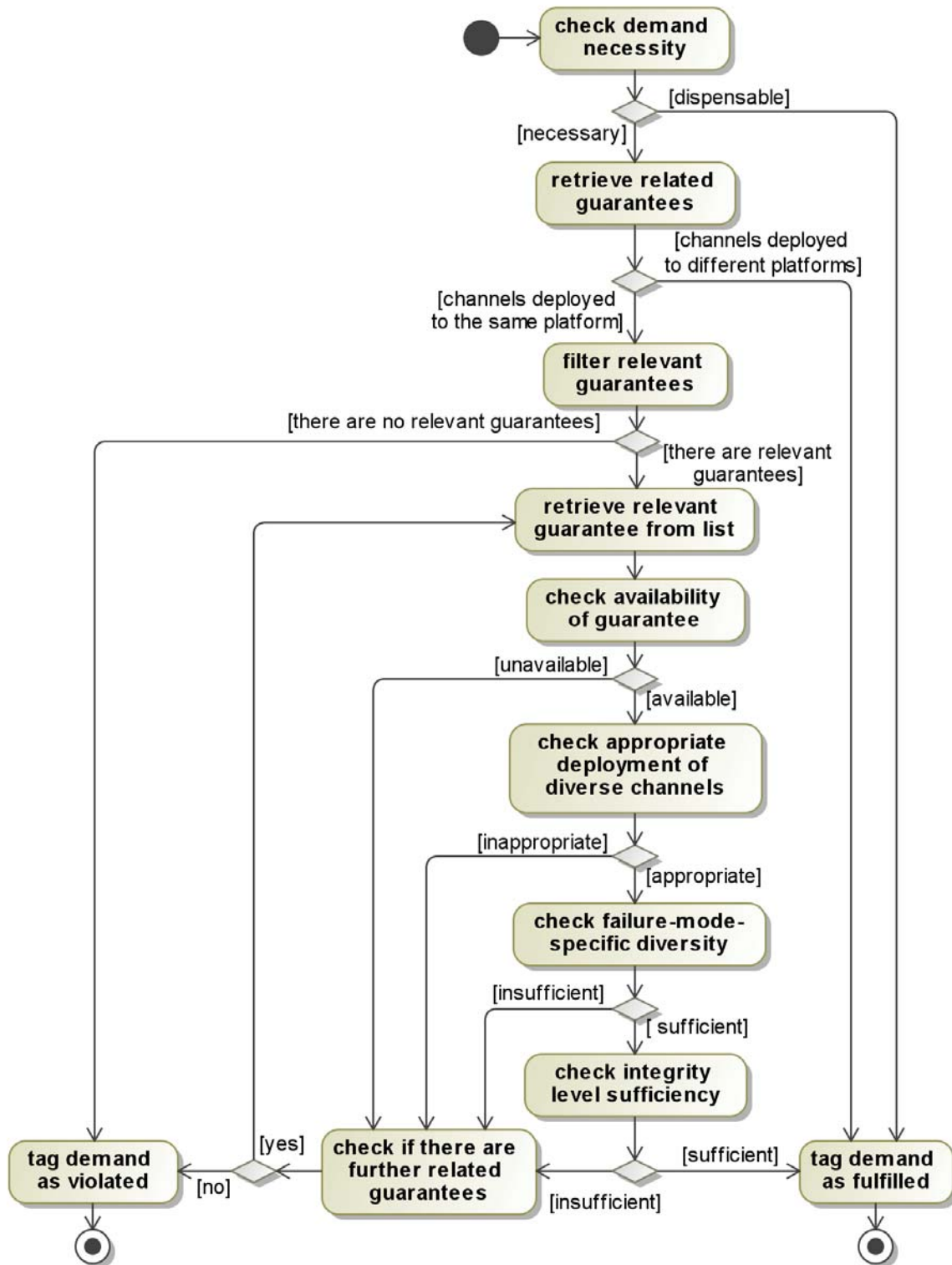


Figure 70: An overview of service diversity demand mediation

5.4.6 Mediation of Specific Parameters

For the detailed specification of failure modes and failure reactions, the VerSal language allows parameterizing certain failure modes and reactions. These parameters influence the mediation of platform service

demands, application monitoring demands, and failure reaction demands as described in the previous sections. In the following paragraphs, we will describe how the mediator treats different kinds of parameters.

Most of the failure modes and failure reactions are parameterized with standard parameter sets that are reused throughout the specification of the VerSal language. These parameter sets are: time deviation, latency, jitter, and error. However, to specify arrival rate failures, analog default signal reactions, and digital default signal reactions, the VerSal language uses specific parameters. These parameters do not, however, influence the mediation.

To find out which parameters are used for the definition of different failure modes, please refer to Appendix B Table 17. To find out which parameters are used for the definition of different failure reactions, please refer to Appendix B Table 18.

Time Deviation

A time deviation parameter models a time constraint specifying the acceptable deviation from a nominal latency. Consequently, this constraint contains two time parameters, the nominal latency (t_n) and the acceptable deviation (t_d) from the nominal latency. For a time deviation constraint, the acceptable time interval t_a results to $(t_n - t_d) < t_a < (t_n + t_d)$.

Nominal latency plays no role in the mediation of a deviation parameter. In case the demand that is parameterized with the time deviation constraint is to be fulfilled with a detection guarantee, the mediator checks whether the smallest detectable deviation is smaller than the acceptable deviation. In case the demand is to be fulfilled by a guarantee, the mediator checks if the avoided deviation promised by the guarantee is smaller than the acceptable deviation specified by the demand.

Latency

The latency constraint allows modeling an acceptable time interval by specifying the acceptable lower bound (t_l) of the latency if an early failure is critical, or the acceptable upper bound (t_u) of the latency if a late failure is critical. In case both early and late failures are critical, the corresponding acceptable time interval t_a therefore results to $t_l < t_a < t_u$.

If either the acceptable lower bound or the acceptable upper bound is not specified, this means that the corresponding failure is not relevant regarding the current demand. Consequently, the mediator omits the related check and only performs the other check. If, however, both

bounds are undefined, the mediation of the latency parameters throws an error and assesses the parameter as violated.

If an early failure is critical, the mediator checks whether the detectable earliest deviation is smaller than or equal to the acceptable earliest deviation, or if the guaranteed lowest deviation is larger than or equal to the acceptable earliest deviation. Analogously, if a late failure is critical, the mediator checks whether the detectable latest deviation is large than or equal to the acceptable latest deviation, or if the guaranteed latest deviation is smaller than or equal to the acceptable latest deviation.

Jitter

Constraints regarding jitters are specified using the *period constraint*. A period constraint models the admissible deviation between two occurrences of a periodical event. A period is defined by its duration (t_n), i.e., by the nominal time between the occurrence of the two subsequent instances of the same periodical event, and by its jitter (t_j), i.e., by the admissible deviation from the nominal duration. The acceptable period p_a between two occurrences of the periodical event is $(t_n - t_j) < p_a < (t_n + t_j)$.

If a period constraint is mediated, the mediator checks whether the detectable jitter is bigger than the acceptable jitter or whether the promised maximum jitter is smaller than the acceptable jitter.

Error

An error parameter describes the admissible deviation of an actual value of a signal from the nominal value of the signal. In terms of mediation, we have to differentiate between relative and absolute errors.

A relative error is specified using an integer parameter that ranges from 0 to 100, modeling the admissible deviation as a percentage. If the error is to be detected, the mediator checks whether the smallest detectable error is smaller than the admissible error. If the error is to be avoided, the mediator checks whether the avoidable error is smaller than the admissible error as well.

An absolute error is specified using a float parameter and a string parameter. The float parameter specifies the admissible error value, whereas the string parameter identifies the unit of the error. If the unit of the demanded error-related capabilities is the same as the unit of the guaranteed error-related capabilities, the mediator continues checking whether the guaranteed value is sufficiently low. However, if the units do not match, the mediator can only regard the parameter as violated since the current version of the VerSal method contains no mechanism for unit conversion.

5.5 Post Mediation

After the mediation is finished, the mediator processes the results of the mediation to provide an overview to the integrator. This overview contains all relevant information regarding the success or failure of the mediation. Based on this information, the integrator decides whether a different configuration, a different integration, or a rework of the manual conditions is capable of transforming a failed mediation into a successful one. Using the mediation results, the integrator is capable of efficiently identifying the causes of a failed mediation, fix the issues, and start a new mediation run.

In the following section, we will introduce the information provided by the mediator after a mediation run.

5.5.1 Visualizing Mediation Results

The mediator visualizes the results of the mediation by generating a separate mediation report for each application that is part of the system and that contains at least one demand. The goal of the mediation report is to explain for every demand why and how it is fulfilled or why the mediator was unable to fulfill the demand. In case the demand is violated, the mediation report should point out what made the mediation fail in order to enable the integrator to quickly fix the problem, if possible.

The mediation report is hierarchically structured. On the top level of the hierarchy, the report contains a list of every interface demand specified by the application. On this level, the mediation result provides a quick overview regarding the fulfillment of the demand, and if the demand is violated, a quick comment regarding the reasons, if possible.

On the second level, the report provides two kinds of information regarding the mediation of the chosen demand. First, it shows the results of the prerequisite checks on the demand level, like demand necessity or the availability of related guarantees. Second, the report lists every guarantee related to the demand, including a quick summary regarding this guarantee's capability of fulfilling the demand.

On the third and last level, the report illustrates the outcome of the various checks performed to assess whether the guarantee at hand is capable of fulfilling the demand. Every prerequisite that has to be checked is listed, including the information regarding the outcome of the check and, in case the guarantee is unable to fulfill the demand, comments that explain why a certain prerequisite is not fulfilled.

With this information, the integrator is not only able to identify which prerequisite check caused a specific demand to be violated, but also to gain additional, prerequisite-internal information via the comment field. With this feature, the mediation report can point the integrator to single conditions and configurations that can change the outcome of the mediation. Figure 71 shows the structure of a mediation report.

Name	Fulfillment	Comment
▶ Platform Service Demand: ASWC_A.ComFailure.Corruption	fulfilled	
▶ Platform Service Demand: ASWC_A.ComFailure.Omission	violated	no related guarantees
▶ ...		
▲ Resource Protection Demand: ASWC_X.Memory	fulfilled	
demand necessity	fulfilled	
related guarantee existence	fulfilled	
protection necessity	fulfilled	
▲ Resource Protection Guarantee: MemoryModule_A	violated	insufficient integrity level
guarantee availability	fulfilled	
integrity level sufficiency	violated	insufficient integrity level (ASIL D > ASIL A)
...		
▶ Resource Protection Guarantee: MemoryModule_B	fulfilled	

Figure 71: The structure of a mediation report

6 Deployment Evaluation

This chapter is the third and final chapter describing our method for “Efficiently Deploying Safety-Critical Applications onto Open Integrated Architectures”. In this particular chapter, we will describe our objective function for evaluating and optimizing the high-level deployment evaluation, which corresponds to the third contribution specified in chapter 1.

Contrib. 3

Deployment Evaluation: *Developing an objective function for evaluating and optimizing the deployment of a functional architecture onto a platform topology from a safety perspective.*

As discussed before, open Integrated Architectures like AUTOSAR or IMA enable flexible deployment, which can potentially help to reduce the number of computer platforms in a distributed embedded system, and therefore reduce weight, energy consumption, and costs. Finding a beneficial deployment that yields the desired properties is, however, a complicated multi-criteria optimization problem. One criterion that requires exceptionally careful examination is safety, since an adverse deployment can compromise system safety and inflict significant costs.

In section 2.2, we identified and listed several safety-related objective functions that assist the integrator in finding a suitable deployment. However, to the best of our knowledge, there is no function that allows the integrator to weigh the costs of strict separation versus flexible deployment in a mixed-critical system. Since mixed-critical systems are gaining more and more importance in the automotive as well as in the aviation industry, our method provides an objective function for evaluating these costs. The objective function is based on two metrics and additional constraints, which are assembled to form an adequate objective function. The assembled objective function is finally implemented and tested using a genetic algorithm (GA).

Figure 72 shows an overview of our deployment optimization contribution as well as its interface to the residual VerSal method. Deployment optimization takes place during deployment planning, before the VerSal method comes into play. Our objective function assists the integrator in finding a suitable deployment plan, i.e., a mapping of ASWCs to platforms and signals to communication links. The generated deployment serves as input to the VerSal method, where the deployment plan is used to configure the applications and platforms accordingly, and as refinement of the deployment plan until the deployment can be realized technically.

The structure of this chapter is as follows. Section 6.1 provides a detailed description of the problem addressed by our approach and introduces a running example that is used to illustrate the working of our metrics and constraints. We present the objective function for deployment evaluation in section 6.2 and the deployment optimization with the GA in section 6.3.

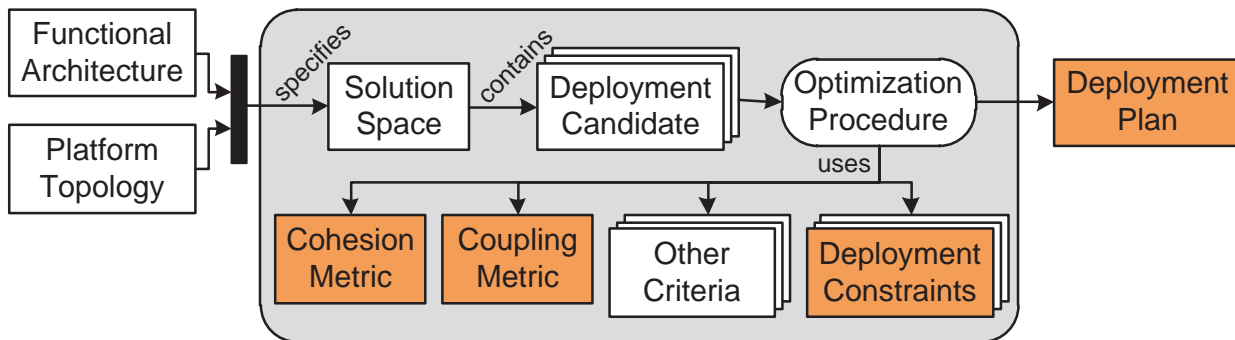


Figure 72: An overview of our deployment evaluation contribution in the context of the overall VerSal method.

6.1 Problem Statement

In an integrated system, applications do not have to be deployed onto platforms as a whole. Applications may consist of several individual ASWCs, which can be deployed separately. To provide a certain degree of separation between ASWCs from different applications, some platforms provide not only one indivisible deployment target, but several individual deployment compartments called partitions. Basically, a partition provides fault containment capabilities such that faults of an application in one partition cannot affect the platform's capability to provide shared resources in such a way that there is an interference with applications in other partitions. Resulting from this definition, we define one aspect of deployment as the mapping of the application software components (ASWCs) onto the partitions of the platforms. ASWCs are further characterized by two attributes: their name and their complexity. Complexity is determined by a three-stage scale (low, medium, high), which provides a very coarse-grained model of the components' implementation complexity and size.

A second aspect of deployment is the mapping of the logical signals exchanged between ASWCs onto the communication channels connecting the different platforms. Here, we differentiate between channels that allow inter-platform communication and the local communication channel that allows communication between ASWCs in different partitions of the same platform. The stronger the separation between interactive ASWCs, the higher the required communication volume.

We specify the target of a deployment as a collection of platforms, where each platform is again capable of containing several partitions. If a platform does not provide partitioning mechanisms, we model this by specifying a platform with one virtual partition. The platforms are connected with each other via communication channels. Platforms and communication channels specify a type that will be used later for the calculation of the objective function. A possible communication channel type could be “CAN” or “FlexRay”, whereas a possible platform type could be “vendor X AUTOSAR 4.0 running on hardware platform Y”.

We label the deployment target, i.e., the tuple consisting of the platform set and the communication channel set, as platform topology. On the other hand, we label the part of the system that has to be deployed, i.e., the ASWCs and the signals, as functional architecture. The goal of deployment optimization is to find a suitable mapping (a deployment plan) of a given functional architecture onto a given platform topology. Figure 73 shows the meta-model we use to specify a deployment problem and possible solution deployment plans.

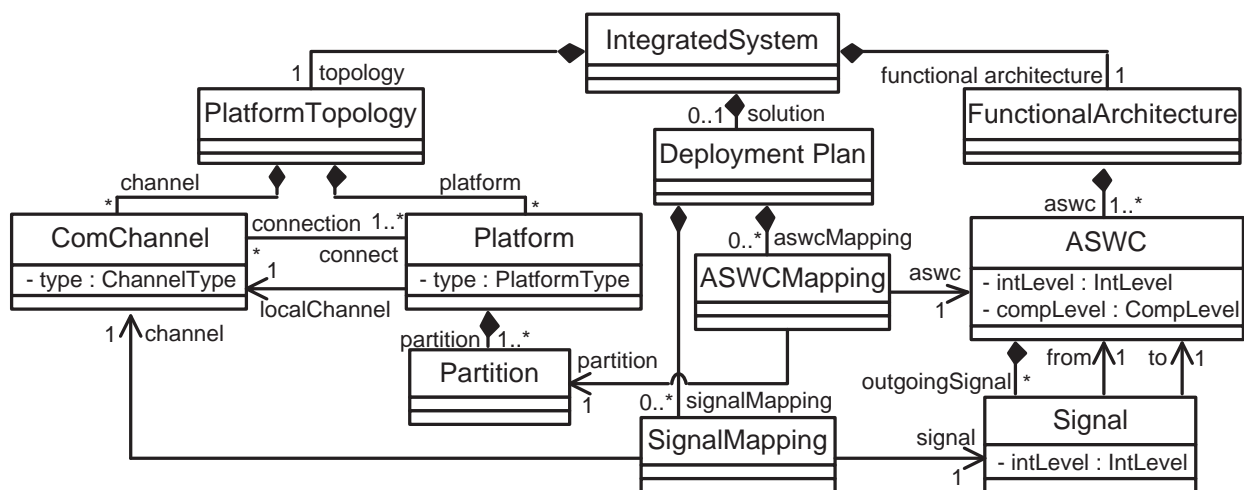


Figure 73: The meta-model to specify deployment problems and possible solution deployment plans.

Please note that the initial deployment planning is performed very early in the development cycle of the embedded system. Since platforms require installation space and the availability of a communication channel affects the wiring, the platform topology affects the geographical design of the embedded system, which is again specified in the early development phases. If it should turn out that the specified platform topology is unable to host all required software components, redesigning the topology becomes expensive. To avoid such a situation, the initial deployment has to be specified early as well in order to show the general feasibility of the platform topology.

During such an early development phase, the deployment is specified on a relative coarse-grained level of detail. It is possible to describe

deployment on a more fine-grained level such that the deployment specifies, for example, the application's requirement on the capabilities of specific platform resources like I/O devices, non-volatile memory (NVRAM), or the assignment of signals to messages or ASWCs to operating system (OS) tasks and so on. Since this information is only available and necessary during the later development phases and our objective function is meant to be used during the early design phase, our objective function does not use this information. However, the more detailed deployment information is safety relevant and has to be checked, which is why we covered these aspects with our VerSal method described in chapter 4 and chapter 5.

6.1.1 Safety-Related Properties

The concept of safety integrity levels (SIL) [59], or comparable concepts like development assurance levels (DAL) [60], is used in safety standards across most domains. Integrity levels are a qualitative scale for the risk posed by a system hazard. The higher the risk, the stronger the requirements for the system to reduce the risk to an acceptable level. Safety standards try to enforce this by tailoring the safety standards using integrity levels. The higher the integrity level, the stronger, the stricter, and the more numerous the demands of the standard. As a consequence, the integrity level significantly influences the development costs of a system. Depending on the criticality level, costs for DO-178C-compliant software [60] development can increase by 300 to 500%.

During system development, while the system architecture is being gradually refined, it is common to allocate integrity levels to components if the safety requirements implemented by that component are required to prevent a hazard that has the corresponding integrity level. Simply tagging a component with an integrity level can be regarded as simplification, as it abstracts from the specific requirement and the specific failure that would actually lead to the hazard. Still, standards specify deployment rules that are based upon integrity levels, where it is common to assign integrity levels (IntLevel) to components, in our case ASWCs.

The same is true for logical signals exchanged between ASWCs. We assign integrity levels to signals if there is at least one failure mode related to the transmission of the signal (like corruption, delay, insertion, masquerading, etc.) that might lead to a hazard that poses the corresponding level of risk.

Since this information is required for calculating our objective function, we assume that the ASWCs and the signals contained in the given functional network are all classified according to their criticality level.

6.1.2 Running Example

Figure 74 shows the specific deployment problem that we will use as a running example to illustrate our deployment evaluation method. The example uses the safety integrity level (SIL) scale defined by the IEC 61508 [59]. The scale goes from QM for uncritical components to SIL 1 - SIL 4, with SIL 4 being the category for the most critical components.

The functional architecture implements a two-channel comparator architecture. Both channels are built of two SIL B ASWCs with medium complexity exchanging two signals. The resulting signal of both channels is then transmitted to the comparator component, which has low complexity but a relatively high SIL C criticality level. In addition to the two-channel comparator, the system contains two highly complex components that implement an uncritical functionality.

The simple platform topology consists of two platforms that both provide a partitioning mechanism. Both platforms provide two partitions. The platforms are connected via one communication channel called CH1.

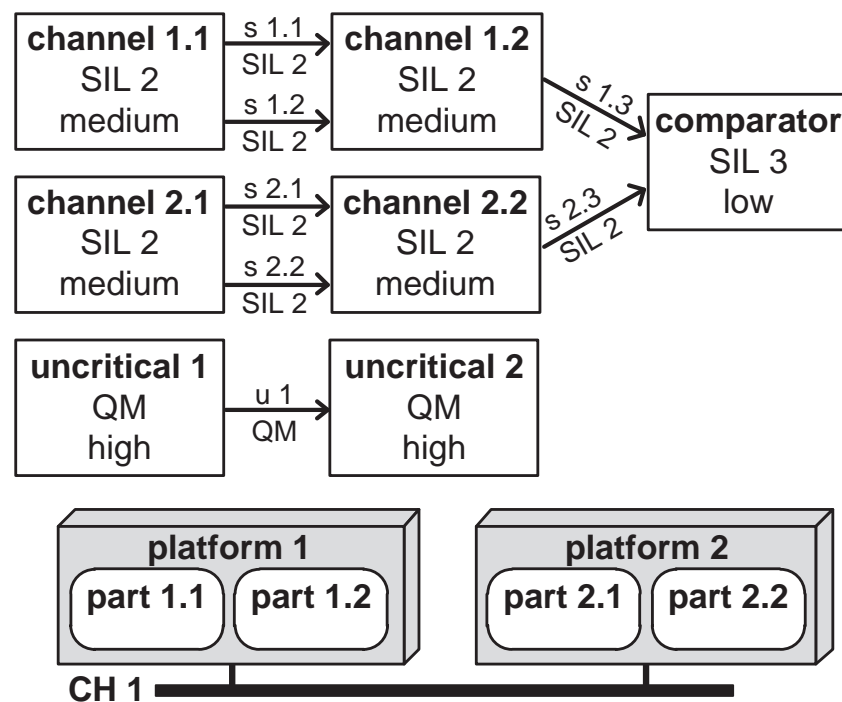


Figure 74:

A running example for the deployment evaluation method . ASWCs are depicted as rectangles containing three strings, from top to bottom: name, criticality, complexity. Signals are depicted as arrows with two strings, from top to bottom: name, criticality.

6.2 Objective Function

This chapter introduces two metrics for evaluating, from the safety perspective, a solution for a specific deployment problem introduced in section 6.1. The metrics implement a cost function that is minimized by the optimization algorithm presented in section 6.3. In particular, both metrics evaluate negative effects caused by two core characteristics of integrated architectures.

The cohesion metric is presented in section 6.2.1 and focuses on the aspect of shared computational resources, as the metric evaluates the costs induced by unprotected interferences between mixed-critical ASWCs. The coupling metric is presented in section 6.2.2 and evaluates the costs caused by safety mechanisms required to protect against communication failures. In addition to the quantitative evaluation using these metrics, we allow for the specification of certain constraints that are required to restrict the available deployment solution space to sensible solutions. These constraints are introduced in section 6.2.3. The assembly of the metrics and the constraints into a single objective function is presented in section 6.2.4. Since the metrics and the objective function are highly configurable, we conclude this chapter with a mechanism to adequately parameterize the metrics in section 6.2.5.

6.2.1 Cohesion Metric

A major disadvantage of integrated architectures is the lack of natural fault containment barriers. If an application fails in a federated architecture, the failure propagates to other applications only via functional dependencies because different applications are hosted on separate platforms, which leaves almost no potential for fault propagation via technical dependencies. However, in an integrated architecture, failures of an application can affect the host platform, and from thereon, affect other applications on the same platform even if the concerned applications share no functional dependencies. This effect is typically called interference.

If there is a possibility that a set of ASWCs will interfere with each other, safety standards typically demand that all ASWCs in the set are developed according to the highest integrity level amongst all ASWCs in the set. This is done to prevent failures of lower-criticality components developed according to less strict development requirements from causing higher-criticality applications to fail and therefore indirectly cause hazards with higher criticality. Conversely, if standards were not to apply this rule, lower-criticality components would be capable of causing highly critical hazards.

In section 6.1.1 we already introduced the concept of partitioning. Partitioning separates a platform into virtual compartments and prevents interferences across the borders of partitions. Inside a partition, however, there is no freedom from interference. As a consequence, ASWCs allocated to the same partition or to a platform that provides no partitioning mechanisms must be developed according to the highest integrity level of all the ASWCS allocated to the partition as described above.

If this rule causes a raise of the original integrity level of an ASWC, development costs increase. The cohesion metric quantifies this effect based on an estimation of the resulting additional costs. According to our experience, the costs for safety-critical development are not added to the regular development costs like a constant, but rather affect the costs like a factor. Therefore, we define $cf_{intLevelA}(x)$ to be the cost factor for the development of an ASWC with integrity level " x ", compared to the development of an identical but uncritical ASWC.

Let x_{org} be the original integrity level of an ASWC and x_{new} the increased integrity level of the ASWC caused by deployment. Then the cost factor difference dcf is calculated as:

$$dcf(x_{org}, x_{new}) = cf_{intLevel}(x_{new}) - cf_{intLevelA}(x_{org})$$

To evaluate the impact of the cost factor difference, we have to estimate the development costs of the affected component. To this end, we define the complexity of an ASWC as a qualitative scale as described in section 6.1, as the costs increase with increasing complexity of the development of the component. We further define the function $cf_{compLevel}(y)$ as the cost factor for complexity level " y ". The complexity categorization of an ASWC is currently based on expert judgment.

If we let $iL(aswc)$ be the integrity level and $cl(aswc)$ the complexity level of the ASWC " $aswc$ ", the cost difference dc for upgrading the criticality of $aswc$ to level x_{new} is defined as:

$$dc(aswc, x_{new}) = dcf(iL(aswc), x_{new}) * cf_{compLevel}(cl(aswc))$$

Finally, the cohesion metric results from summing up the cost differences for all applications in all partitions. Let " P " be the set of all partitions of the platform topology and $max_{intLevel}(part)$ the maximum integrity level among the applications in the partition " $part$ ". Then cohesion is calculated as:

$$coh(P) = \sum_{part \in P} \sum_{aswc \in part} dc(aswc, max_{intLevel}(part))$$

Figure 75 shows two solutions for deploying the running example side by side. The solution on the left shows a deployment yielding no cohesion costs, since there are only equally critical ASWCs in each partition. The deployment shown on the right yields much worse cohesion since both uncritical complex components are deployed to the same partition as the critical comparator component. This would result in two highly complex but uncritical components being developed according to SIL C.

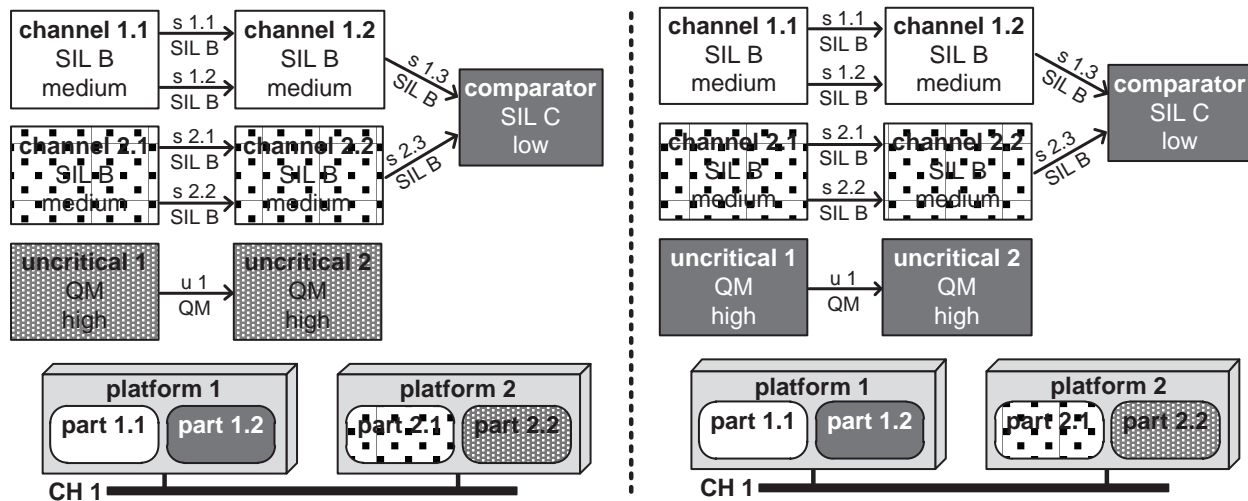


Figure 75: Two example deployments illustrating the cohesion metric. The deployment of an ASWC to a partition is indicated by the same fill color and pattern of the respective shapes. The deployment of signals is not indicated.

6.2.2 Coupling Metric

In an integrated architecture, computer platforms are interconnected via communication buses. This allows the system developer to spread the components of an application over multiple platforms and to integrate applications in order to provide new or improved functionalities. However, the increased information exchange caused by spreading an application over multiple platforms is also an additional source of failure.

In a safety-critical system, communication failures can potentially cause hazards, which is why protection mechanisms are necessary to detect and control them. Typical protection mechanisms include sending redundant information to detect corruptions, message counters to detect lost messages, or deadline monitoring to detect delayed signals. These mechanisms cause bus workload, use computational resources, and may also increase end-to-end delay. Furthermore, communication protection mechanisms typically detect, but do not prevent failures. The necessary failure reaction often lowers the utility or availability of the system. Therefore, the coupling metric evaluates these costs of safety-critical communication.

In section 6.1.1, we abstracted from specific communication failure modes and classified each signal by assigning an integrity level to it. With increasing risk, standards typically demand increasingly rigorous protection mechanisms. To achieve high diagnostic coverage, for example, ISO 26262 recommends complete bus redundancy, whereas multiple redundant bits optimally allow for medium diagnostic coverage. To represent this, we evaluate the costs for protecting a signal as a function $cf_{intLevels}(x)$ of the signal's integrity level " x ".

The costs for protecting signals from communication failures do not solely depend on integrity levels. They also depend on the communication channel that the signal is transmitted on. This is because some types of channels already come with protection mechanisms or have a design that makes certain failures less likely. In this dissertation, we only differentiate between intra-platform communication (if the collaborating ASWCs are located in different partitions of the same platform) and inter-platform communication (if they are located on different platforms). Channel-type-specific costs can be differentiated further by adding more channel types to the meta-model and extending the function $cf_{channelType}(cT)$, which yields the cost factor of a channel type cT .

If we let " s " be the evaluated signal, $cT(s)$ the type of the communication channel that " s " is assigned to, and $iL(s)$ the integrity level of " s ", then the cost function for protecting the communication of " s " is defined as:

$$cc(s) = cf_{channelType}(cT(s)) * cf_{intLevels}(iL(s))$$

If we let $aswcNet$ be the set containing all applications and $oS(aswc)$ the outgoing signals of the ASWC " $aswc$ ", then coupling is defined as:

$$coup(aswcNet) = \sum_{aswc \in aswcNet} \sum_{s \in oS(aswc)} cc(s)$$

Figure 76 shows two solutions for deploying the running example side by side. The deployment shown on the left side yields low coupling costs since only the signal "**s 2.3**" with SIL B criticality is deployed to an inter-platform channel. The second signal that requires communication is the signal "**s 1.3**", but this signal can be transmitted at lower costs via the platform-internal communication channel. The deployment shown on the right side, however, requires the inter-platform communication of five additional signals, which results in much higher coupling costs.

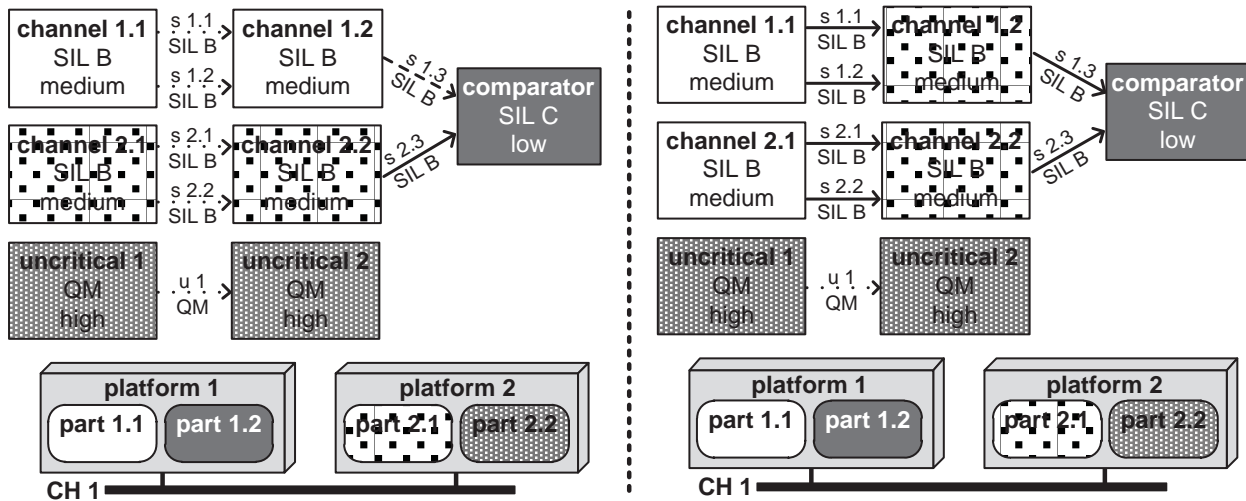


Figure 76: Two example deployments illustrating the coupling metric. The deployment of ASWCs is indicated as in Figure 75. The deployment of a signal is indicated as follows: Locally exchanged signals are shown with a dotted line. Signals deployed to the respective intra-platform channel are shown with a dashed line. Signals deployed to the inter-platform channel “CH 1” are shown with a solid line.

6.2.3 Constraints

This section introduces two constraints that allow the designer to restrict the deployment solution space. Whereas the aspects evaluated in the previous two sections have quantifiable effects on system development, solutions that violate constraints are infeasible and will therefore be discarded.

We mentioned before that our metrics operate during very early development stages, where most of the software has not been developed yet. This assumption is used for calculating the cohesion metric, where we assume that we can still change the native integrity level of components. However, if the component is reused from a previous project or if it is a component off the shelf (COTS), the component is already developed and its integrity level can no longer be freely adapted. Therefore, the first constraint allows the designer to specify that a certain ASWC has a fixed integrity level. Consequently, it will be treated as a constraint that the ASWC does not share a partition with lower-criticality ASWCs.

The second constraint is used to represent dissimilarity relations between typically two or three ASWCs, which means that the corresponding ASWCs have to be developed heterogeneously to avoid systematic common-cause failures. This also means that the platforms the ASWCs are deployed to must not have systematic common-cause failures either. Consequently, the dissimilarity constraint is violated if the type of the host platforms of at least two dissimilar ASWCs is the same.

6.2.4 Objective Function Assembly

Since we intend to use a GA to evaluate our objective functions and GAs typically work with fitness functions, we have to transform the cost functions implemented by our metrics into a fitness function. To that end, we define a function to pessimistically estimate the worst-case costs for a specific deployment problem and subtract the cost functions to get a non-negative fitness function. The pessimistic worst-case estimation simply assumes that every ASWC is deployed onto the same partition and every signal is transmitted via an inter-platform communication link.

Let "*sdp*" be a specific deployment problem (a tuple consisting of a platform topology and a functional network). Then *parts(sdp)* yields all partitions of all platforms, and *aswcs(sdp)* yields all ASWCs in *sdp*. Furthermore, let *wce(sdp)* be defined as the corresponding worst-case cost estimation and *const(sdp)* as a function that yields "1" if no constraint is violated and "0" if at least one constraint is violated. Then we define the fitness function as:

$$fit(sdp) = const(sdp) * (wce(sdp) - coh(parts(sdp)) - coup(aswcs(sdp)))$$

Using a number of exemplary architectures and corresponding deployments, we conducted a qualitative analysis of the fitness function with practitioners in the automotive domain. During the analysis, several iterations were necessary to adapt the problem description and the metrics such that it became possible to model the relevant aspects of the deployment and to evaluate them appropriately. After a final evaluation of the technique, the metrics were identified as adequately expressive and the expert estimations allowed for adequate parametrization of the metrics.

6.2.5 Parameterization

Our objective function requires adequate parameterization to function properly. If we assume three complexity levels, two kinds of communication channels (inter- and intra-platform channels), and the common number of five criticality levels (including the uncritical level) for ASWCs as well as for signals, we end up with a total of fifteen parameters for customizing the cost functions as shown in Table 10.

Table 10:

A list of parameters of our objective function

Number	Parameter	Description
1	$cf_{compLevel}(low)$	Cost factor for a ASWC with a low complexity
2	$cf_{compLevel}(medium)$	Cost factor for a ASWC with a medium complexity
3	$cf_{compLevel}(high)$	Cost factor for a ASWC with a high complexity
4	$cf_{intLevelA}(QM)$	Cost factor for a ASWC with a QM risk estimation
5	$cf_{intLevelA}(SIL_1)$	Cost factor for a ASWC with a SIL 1 risk estimation
6	$cf_{intLevelA}(SIL_2)$	Cost factor for a ASWC with a SIL 2 risk estimation
7	$cf_{intLevelA}(SIL_3)$	Cost factor for a ASWC with a SIL 3 risk estimation
8	$cf_{intLevelA}(SIL_4)$	Cost factor for a ASWC with a SIL 4 risk estimation
9	$cf_{channelType}(intraP)$	Cost factor for a ASWC with a medium complexity
10	$cf_{channelType}(interP)$	Cost factor for a ASWC with a high complexity
11	$cf_{intLevels}(QM)$	Cost factor for a ASWC with a QM risk estimation
12	$cf_{intLevels}(SIL_1)$	Cost factor for a ASWC with a SIL 1 risk estimation
13	$cf_{intLevels}(SIL_2)$	Cost factor for a ASWC with a SIL 2 risk estimation
14	$cf_{intLevels}(SIL_3)$	Cost factor for a ASWC with a SIL 3 risk estimation
15	$cf_{intLevels}(SIL_4)$	Cost factor for a ASWC with a SIL 4 risk estimation

We chose to design this flexibly since an exact acquisition of specific safety-related costs is usually not available and different domains, organizations, and sometimes different projects will most probably require different kinds of parameterization regarding safety-related costs.

Since it is usually difficult for the developers to acquire the respective cost relations, we allow for an alternative way to parameterize the metrics: A deployment expert is confronted with an artificial but humanly manageable calibration deployment problem. The deployment expert is allowed to change the parameters, and after each change, the optimizer immediately calculates a deployment solution and presents it to the expert. This cycle is repeated until the optimizer arrives at a solution that the expert expects.

The parameter set that produced the expected solution during the calibration can then be used for real-world deployment problems. Please note that the quality of the resulting parameters depends on the expert's estimation and might not correlate with the real cost factors. According to our experience, however, this process yields better parameters than completely manual parameterization. Figure 77 illustrates this tool-supported parameterization process.

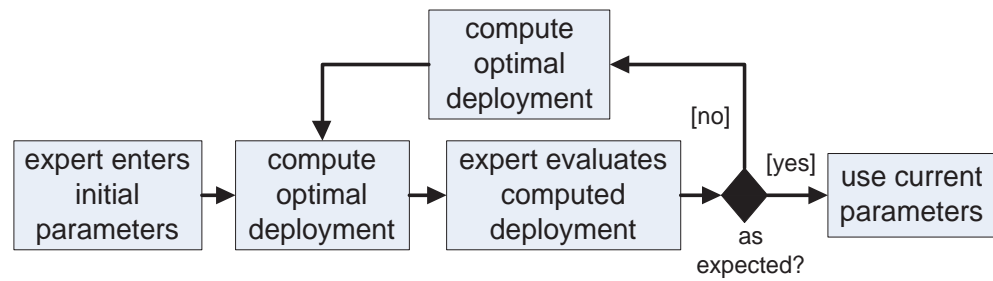


Figure 77: The tool-supported parameterization of the objective function

6.3 Deployment Optimization

In this section, we present a deployment optimization algorithm based on the introduced fitness function and a GA. It is important to note that the focus of our work lies on the presented metrics and not on the selection of this specific optimization algorithm. We used a GA to test and evaluate our metrics because they were integrated into a larger-scale optimization running a GA as well. Other techniques, such as linear programming, however, are also suitable for deployment optimization.

A GA is a stochastic search algorithm that uses techniques adopted from natural evolution to find near-optimal solutions for complex optimization problems [69]. The optimization process starts with a number of randomized solutions, the so-called initial population. After initialization, each member of the population is evaluated for its fitness, and then a new population is reproduced from the old population using techniques like crossover, where chromosomes of one individual are mixed with chromosomes of another individual, and mutation, where single chromosomes are randomly altered. Members with higher fitness are more likely to participate in this reproduction/generation of a new

population than members with low fitness. After the new population is generated, it is evaluated for its fitness, which is followed by another reproduction of a new population. This optimization loop terminates, for example, after a fixed number of cycles or when one individual has reached a sufficient predefined fitness.

To be able to use standard algorithms like crossover and mutation, solutions of the optimization problem have to be represented by so-called chromosomes. A chromosome is divided into several genes, each gene representing a distinct part of a potential solution. In our case, the intuitive chromosome for a specific deployment problem with k ASWCs and n signals would be an array of k genes representing ASWC mappings, concatenated with n genes representing signal mappings.

However, we decided to include only the ASWC mappings onto the chromosome and let the GA optimize only the ASWC mappings. This is because the signal mapping highly depends on the ASWC mapping and we are able to calculate the optimal signal mapping directly as soon as the ASWC mapping has been determined. For a specific deployment problem with k ASWCs and m partitions, our chromosome therefore consists of the genes $g_j \in \{1, \dots, m\}, j \in \{1, \dots, k\}$. Each of the k genes is represented by an integer between 1 and m , where $g_a = b$ denotes that ASWC a is assigned to partition b .

Using this chromosome layout results in a slightly adapted version of the aforementioned GA optimization loop, since we have to add the signal mappings to the ASWC mappings for calculating our fitness function. The resulting loop consists of three steps: (1) calculate the fitness for each individual, (2) reproduce a new set of ASWC mappings, (3) calculate optimal signal mappings for each individual. The optimization stops if the fitness improvement within the last 30 generations has been below 5%. Figure 78 shows the modified optimization loop.

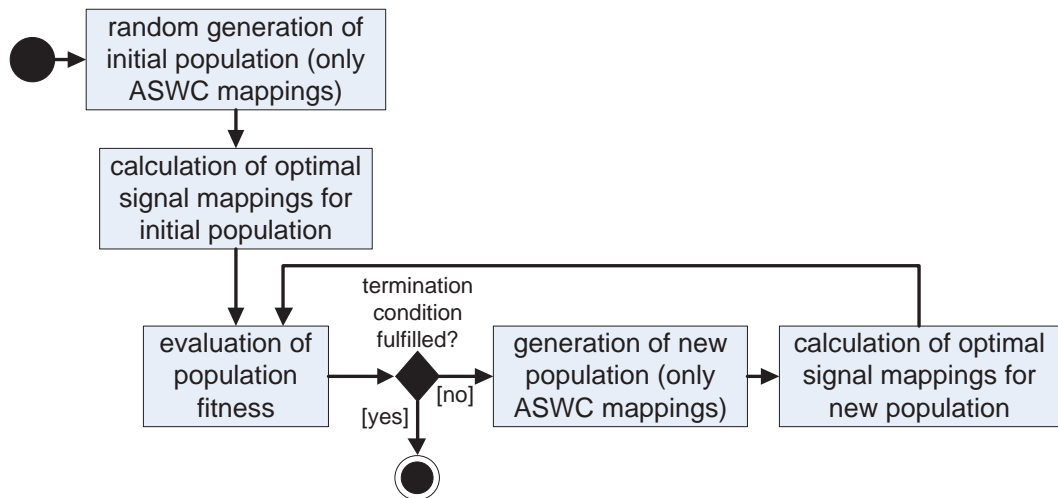


Figure 78: The adapted GA optimization loop used for our objective function

The optimal signal mapping can be determined in a straightforward fashion since the costs for individual signal mappings do not influence each other. First, we check the deployment of the receiver and sender ASWC of each signal. If both are in the same partition, no channel is needed, and if both are on the same platform, we deploy the signal to the local channel. If both are hosted on different platforms, we search for all available channels connecting the respective platforms. If there is no such channel, we flag the ASWC mapping as invalid. If there is more than one channel, we search for the channel that yields the lowest costs and deploy the signal accordingly.

7 Implementation and Evaluation

In this chapter, we will describe the evaluation and the technical implementation of our contributions described in the preceding chapters. Comparable to the structure of the overall thesis, we performed the evaluation in a modular way: In the first part of the evaluation, we will assess the VerSal technique, which includes the VerSal language as well as the VerSal mediator. The VerSal language was specified earlier, in chapter 4, whereas the specification of the VerSal mediator is found in chapter 5. In the second part of the evaluation, we will assess our objective function for deployment evaluation, which was specified in chapter 6. The technical feasibility of our solution is underpinned by two implementation prototypes that will be introduced in the following as well.

This chapter is structured as follows. The evaluation of the VerSal technique is described in section 7.1; the evaluation of our approach for deployment evaluation is described in section 7.2. The implemented prototypes of the VerSal technique and the deployment evaluation are introduced in section 7.2 and section 7.4, respectively.

7.1 VerSal Evaluation

The VerSal technique comprises mainly two components. First, there is the VerSal language, which allows application and platform developers to specify the demands regarding the safety-related behavior of the platform and the correspondingly safety-related guarantees of the platform. This part of the VerSal technique corresponds to our first contribution, in which we declared that we will *“define a formal language for the modular specification of safety-related demands and guarantees between an application and a platform in an open integrated architecture.”* Second, there is the VerSal mediator for automatically checking if a set of application demands specified with the VerSal language is fulfilled given the guarantees provided by a specific platform. This part of the VerSal technique corresponds to our second contribution, where we proclaim that we will *“develop an automated process for checking the safety compatibility of an application and a platform in an open integrated architecture”*.

In this section, we evaluate whether both of these contributions have been achieved by the solutions presented in the previous chapters, i.e., whether it is actually possible to specify the relevant demands and guarantees with the VerSal language and whether the mediator is

capable of automatically checking the compatibility of demand and guarantee interfaces specified with our language.

Our evaluation starts with the argument for the applicability of the VerSal language. The basic strategy behind this argument is to show that the VerSal language is capable of covering the safety-related dependencies between a state-of-the-practice application and a state-of-the-practice platform. This state of the practice in the development of open integrated architectures is well represented by the most widely used open integrated architectures, which are AUTOSAR in the automotive domain and the civil IMA-derivate ARINC 653 in the aviation domain. Consequently, to demonstrate that the VerSal language is capable of covering the relevant safety-related dependencies, we refer to the most widely used open integrated architecture standards: AUTOSAR and ARINC 653.

To show that our language covers AUTOSAR and ARINC 653, we generated a mapping between these standards and the VerSal language. As described in section 4.2, the VerSal language is structured into four classes/packages of safety-related dependencies between applications and platforms: (1) platform service failures, (2) health monitoring, (3) resource protection, and (4) service diversity. In the following paragraphs, we will iterate through each of these classes and discuss their completeness with regard to AUTOSAR and ARINC 653 specification.

The first step in arguing the completeness of the *platform service failures* covered by the VerSal language is to argue the completeness of the platform services covered by the VerSal language. These are: synchronization, communication, I/O access, time services, memory services, and scheduling. In the following tables, we map the services provided by ARINC 653 (see Table 11) and AUTOSAR (see Table 12) to the VerSal services. Please note that the mapping tables include the mapping of “health monitoring” services. Health monitoring is not relevant for the platform service failure class and will be discussed separately shortly hereafter.

Table 11: Mapping of AUTOSAR services to services in VerSal . An “x” in the matrix denotes that the specific part of the ARINC API is addressed by the respective service class. Please note that there are further directly accessible service components but those components are only active during initialization or debugging and are therefore not included.

AUTOSAR \ Classification	Com	NV- RAM Mngr.	OS	Diagn. Event Mngr.	Funct. Inhi. Mngr.	RTE	Watch -dog Mngr	IO HW Abstr.
Synchronization Mechanisms			x			x		
Communication	x		x			x		

I/O Access						X
Time Services		x				
NV Memory Access	x					
Scheduling		x			x	
Health Monitoring			x	x		x

Table 12: Mapping of ARINC 653 services to services in VerSal . An “x” in the matrix denotes that the specific part of the ARINC API is addressed by the respective service class.

ARINC 653 \ Classification	Partition Mgmt.	Process Mgmt.	Time Mgmt.	Inter-Part Com	Intra-Part Com	Health Mgmt.
Synchronization Mechanisms				x		
Communication				x	x	
I/O Access	is not directly covered by ARINC 653					
Time Services			x			
NV Memory Access	is covered by ARINC 653 Part 2					
Scheduling	x	x	scheduling implementation by OS			
Health Monitoring	x	x				x

As the tables show, the standard services are well covered by the VerSal approach. Based on these standard services, we identified the failure modes included in the VerSal language by performing a state-of-the-art safety analysis. The analysis was conducted using a guide-word-driven process widely used in academia and industry (see section 4.4.1 for more information regarding the analysis technique). Where possible, we cross-checked the results of our failure analysis with existing failure models specified in the related safety standards (e.g., IEC 61508-2 Table A.1 [59] and ISO 26262-5 Table D.1 [46]) to further align our failure models with existing and accepted failure models. Of course, it is almost impossible to identify a complete set of failure-modes for such a complicated set of services. However, through various iterations in the SPES2020 and ARAMiS project as well as through discussions with industrial partners, we believe that we have reached a stable set of failure-modes for the covered services. However, should the VerSal language not cover a

certain aspect, the language can be extended as we discuss later in this section.

The second dependency class, the *health monitoring* class, allows the VerSal user to specify demands and guarantees regarding mechanisms for both application monitoring and failure control reactions provided by a standard execution platform.

The completeness of the available application monitoring mechanisms with regard to the application monitoring mechanisms available in AUTOSAR and ARINC 653 is shown in Table 13, where we map the VerSal mechanisms to the AUTOSAR and ARINC 653 mechanisms. If the reader is familiar with AUTOSAR or ARINC 653, he or she might notice that the corresponding standards contain additional monitoring mechanisms that are not listed in the table. These mechanisms, like ISR disable budget monitoring in AUTOSAR or illegal OS service call monitoring in ARINC 653, serve the protection of other applications running on the same platform and do not assist the application safety concept of the causative application. Therefore, these mechanisms are not listed as application monitoring mechanisms in VerSal but are covered by the resource protection class.

Table 13: Mapping of monitoring mechanisms to the VerSal language . This table lists the different application monitoring mechanisms provided by AUTOSAR and ARINC 653 and maps them to the platform failure reaction provided by the VerSal health monitoring package (see section 4.4.3)

AUTOSAR OS	AUTOSAR Watchdog Manager	ARINC 653 Health Monitoring	VerSal Health Monitoring
execution time			execution time
inter-arrival time			inter-arrival time
	alive		inter-arrival rate
	sequence		logical sequence
	deadline	deadline	deadline

The completeness of the available platform failure reactions is shown in Table 14. Please note that some mechanisms are labeled differently in AUTOSAR, ARINC, and our technique. Please note further that the VerSal language is not capable of differentiating between shutting down the OS, restarting the MCU, and restarting the complete platform via watchdog reset. Since this is a very AUTOSAR-specific differentiation we chose to abstract and map them to the **platform restart** reaction.

Table 14:

Mapping of platform failure reactions to the VerSal language : This table lists the different platform failure reactions provided by AUTOSAR and ARINC 653 and maps them to the platform failure reaction provided by the VerSal health monitoring package (see section 4.4.4). Acronyms in the table: FIM stands for Function Inhibition Manager. DEM stands for Dagnostic Event Manager.

AUTOSAR OS	AUTOSAR Watchdog Manager	AUTOSAR FIM & DEM	ARINC 653 Health Monitoring	VerSal Health Monitoring
shutdown OS			stop module	shutdown platform
			restart module	restart platform
terminate partition	terminate partition		stop partition	shutdown partition
restart partition			restart partition	restart partition
terminate task			stop process	shutdown task
restart task			restart process	restart task
call error hook			user callback	handler execution
	MCU reset			shutdown platform
	watchdog reset			shutdown platform
	indication	(rule-based) indication		indication

The *resource protection* package provided by the VerSal language is the third of four dependency classes and allows specifying demands and guarantees regarding the protection of shared resources in order to prevent inadvertent interferences between different application components. The structure and completeness of this dependency class were evaluated in an industrial context together with a tier-1 automotive supplier. During a joint project we analyzed an example multicore platform running AUTOSAR regarding potential interferences and available protection mechanisms. Among other aspects, the analysis included almost every AUTOSAR service directly accessible from the application level (NVRAM Manager, Communication Manager,

Diagnostic Event Manager, Function Inhibition Manager, Watchdog Manager, and BSW Mode Manager), as well as certain peripherals (ADC, GPIO, timer, CAN controller, FlexRay controller, and DMA controller) and the service, memory, and timing protection provided by the AUTOSAR OS. The protection demands and guarantees resulting from the interference analysis were used to refine and validate the soundness of the resource protection package.

The final dependency class covered by the VerSal language is called *service diversity*. This class allows the user to demand or guarantee the availability of diversely developed services to support a heterogeneous redundancy concept on the application level. This class of dependency represents a relatively special and rarely used case. Therefore, we did not design it for completeness but rather so that it is suitable for covering the relevant safety architectures that require this dependency class (e.g., certain implementations of the standardized e-gas safety concept from the automotive domain). Should the advancement of integrated architectures cause these architectures to be used more often, we need to extend and refine this dependency class.

In the previous paragraphs, we argued that the VerSal language covers the standardized services specified in AUTOSAR and ARINC 653. However, both standards further allow for non-standardized interfaces between applications and platforms (see complex device drivers in AUTOSAR or system partitions in ARINC 653). These non-standardized interfaces allow the developer to introduce application-specific features into the platform. Since these services are customary in nature, they are not covered by the VerSal language. However, both standards discourage the use of these features as they impair application portability. Yet, should the user require a specific custom service more frequently, the VerSal language can be extended to cover the service.

Now that we have argued the attainment of our first contribution by showing that the VerSal language covers the relevant parts of the AUTOSAR and ARINC 653 standards, our evaluation continues with the second contribution, the automated mediation of VerSal interfaces. In order to demonstrate that the concepts and algorithms for interface mediation presented in chapter 5 can be implemented to produce an automatic mediation tool, we developed a tool prototype. This prototype was implemented based on Java, Eclipse, and the Eclipse modeling framework. The prototype will be introduced in more detail in section 7.3.

Our VerSal prototype was structurally tested with various example inputs to evaluate the correctness of the proposed algorithms. The test set was designed to achieve decision coverage of each of our five mediation algorithms presented in section 5.4.1 to section 5.4.5. The reader can best follow up on the various decision points involved in each algorithm

via the activity diagrams shown for each algorithm. In total, we required 54 test runs to achieve decision coverage as listed in Table 15. Naturally, we found many software bugs during testing. Yet, more interestingly, during the implementation of the algorithms we found several cases in which we were not capable of retrieving information via the meta-model as initially assumed during conception. Mostly this was because of missing references that did not allow us to traverse the model as intended. As a consequence, we had to adjust the meta-model in more than one case.

Table 15:

An overview of the VerSal testing : This table lists the different test runs required to reach decision coverage for the individual mediation algorithms. The table also provides a link to the chapter where the corresponding mediation algorithm is described as well as a link to the figure showing the activity diagram depicting the algorithm.

<i>Mediation Algorithm</i>	<i>Chapter</i>	<i>Figure</i>	<i>Decisions#</i>	<i>Test Runs#</i>
Platform Service Demand Mediation	5.4.1	Figure 65	10	11
Application Monitoring Demand Mediation	5.4.2	Figure 67	9	10
Reaction Demand Mediation	5.4.3	Figure 68	12	13
Resource Protection Demand Mediation	5.4.4	Figure 69	10	11
Service Diversity Demand Mediation	5.4.5	Figure 70	8	9

We discussed the VerSal technique in the context of several industrial and research projects. Our approach was generally perceived as helpful, but it was criticized for having solely model-based representation of demands and guarantees, which does not fit into the current development life-cycle of a safety-critical system.

In the development life-cycle of a safety-critical system, safety-related demands and guarantees are represented by safety requirements. A demand specifies the need for the implementation of a safety requirement requested by a component that is otherwise unable to execute safely. A guarantee, on the other hand, specifies that a specific safety requirement has been successfully implemented. In the ideal case, the model-based specification of the requirement with the VerSal language replaces the manual specification of the requirement entirely. Yet, since developers and assessors require a natural-language representation to read and understand the safety requirements, the semi-formal specification of the requirement using VerSal has to be

additional in nature, and cannot replace the natural language specification.

In order to circumvent the expensive redundant specification of model-based safety requirements, we evaluated an automatic import/export function based on a structured-text-based approach known from requirements engineering. In a prototypical implementation that included approximately 40% of all demands and guarantees specifiable with the VerSal language, we showed that the demands and guarantees can be transformed from a natural language representation into their model-based VerSal representation and vice versa. For the implementation of the import function we used EBNF (Extended Backus-Naur Form) rules and a structured text representation for the safety requirements. Some example EBNF rules created during the evaluation are shown in Appendix C. Since a VerSal interface specification can be imported automatically from a natural language specification, the additional effort for generating a VerSal interface can be reduced.

However, there are certain restrictions for the application of the VerSal approach: First, the specification of the demands and guarantees in natural language has to follow the rules of predefined structured text templates and the engineer cannot specify them freely. Second, to specify a demand or a guarantee using the VerSal language, and to import it, too, there has to be a model-based representation of the applications and the platforms involved in the system. For our approach we assume that these models are already specified as a part of the regular state-of-the-art engineering process, as, for example, demanded by the AUTOSAR development methodology. Yet, if these models have to be generated in an additional work step, the efficiency of the VerSal technique is reduced.

7.2 Deployment Evaluation

The VerSal technique takes the deployment specification, i.e., the mapping of the application to execution platforms, as input to match application demands with their corresponding platform guarantees. To assist the automated derivation of a suitable deployment plan, we specified an objective function in chapter 6 to be used as a module in a multi-criteria deployment optimization. The development of this objective function corresponds to the third and last contribution of this thesis, in which we stated our goal as *“developing an objective function for evaluating and optimizing the deployment of a functional architecture onto a platform topology from a safety perspective.”*

More precisely, the aim of our objective function is to evaluate the additional costs of mapping mixed-critical functions into an open integrated system. Such additional costs can either accumulate by not

strictly partitioning software components with different criticality levels, or by additional communication protection mechanisms resulting from the distribution of safety-critical functions over the nodes of the platform topology. In this section, we will evaluate whether our objective function produces valid assessments of a given deployment.

Our objective function was evaluated in an industrial context together with experts of a major tier-1 automotive supplier. The evaluation was performed in a two-stage process. In the first phase of the evaluation, we assessed the objective function in an “open-loop” setting that did not involve an iterative optimization; in the second phase, we integrated and evaluated the objective function in an optimization setting that used a genetic algorithm.

The “open loop” optimization was performed using an early prototype that required the user to manually specify the deployment of a given function network onto a give platform topology. Together with a set of configuration parameters, the objective function would take the specified deployment, calculate the costs of the given deployment, and provide it as output to the user. Using this early prototype, the user was capable of specifying several deployments, let the prototype calculate the costs for each deployment, and compare the results of the objective function to the expected results. This first open-loop evaluation of the objective function is depicted in Figure 79.

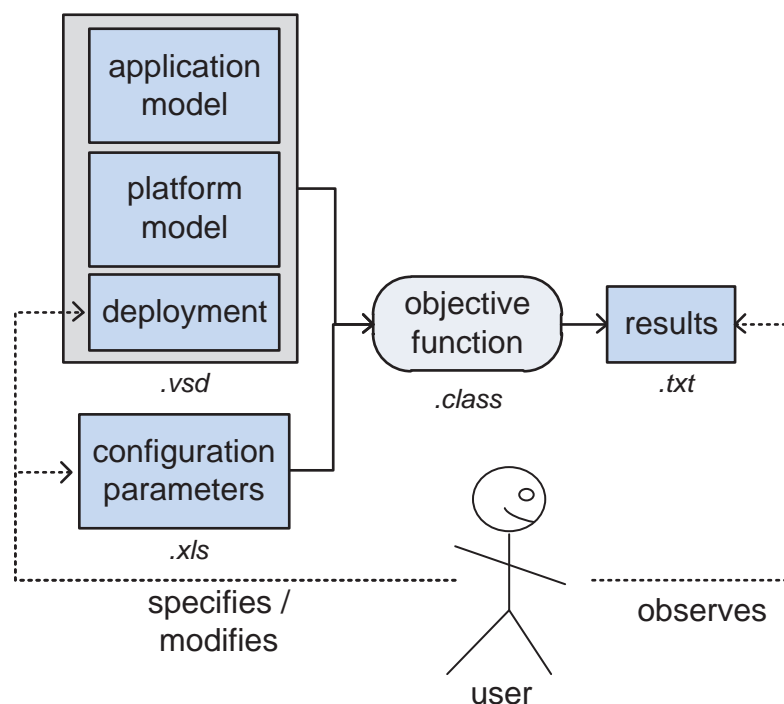


Figure 79: The open-loop evaluation of the objective function using our first prototype

We used the open-loop set-up to focus the evaluation on the objective function itself. The evaluation was performed by an expert from the tier-1 automotive supplier mentioned above. In the first step, the expert specified a simple function network and platform topology. In the second step, the expert specified several deployments and evaluated each deployment manually to compare the expert judgment with the assessment results produced by our objective function. This evaluation revealed two major weaknesses of that early version of the objective function.

The first weakness was related to the addition of both metrics (see section 6.2.1 and section 6.2.2), which yields the result of our objective function. This direct addition required the user to weight both metrics carefully against each other, so that one metric would not superimpose heavily on the other metric. Finding a good weight was very tedious, since both metrics were influenced by a number of parameters that all required simultaneous adjustment. We addressed this issue by introducing an overall weight parameter that allowed the specification of an amplification factor for a metric as introduced in chapter 6.

The second weakness was based on the cost calculation for protecting safety-critical signals sent via communication links. In the original version of the metric, costs would increase linearly with each additional signal. In reality, however, certain costs (e.g., the acquisition or development of a high integrity com stack) are only incurred once, whereas other costs are incurred for every signal. As a result, we modified the cost function to allow the user to specify that certain costs are only counted for the first but not for every following signal that is transmitted via a communication link.

After this first validation of the objective function, we developed a second tool prototype that allowed us to evaluate the objective function in the loop with an optimization algorithm. Since our industrial partner was already evaluating genetic algorithms at that time, we selected genetic algorithms as well. Unlike the previous evaluation phase, which focused on the cost assessments produced by our objective function, this second tool prototype directly provided the user with an optimized deployment. This second evaluation set-up is shown in Figure 80.

Since our second and final prototype produced a deployment, we were only able to judge the quality of the evaluation algorithm indirectly, i.e., based on the quality of the resulting deployment. However, the focus of this second evaluation phase was not directly on the evaluation of the objective function but on evaluating whether the objective function could be used to optimize a real-world example.

The platform topology of our real-world example (taken from [78]) used for evaluation was a replication of a power train platform topology of a

high-end vehicle consisting of 13 platforms connected via FlexRay and CAN (some platforms were connected to both CAN and FlexRay). The function network on the other side consisted of several mock-up functions that were generated to the likeness of an example cruise control application provided by our industrial partner. In order to test our objective function with more than one example application, we took the cruise control and modified it several times to yield comparable applications.

The resulting function network comprised three applications consisting of 27 ASWCs that exchanged 51 signals. According to the judgment of our industrial partner, the optimizer calculated valid deployments for this real-world example. However, even though the genetic algorithm generated valid deployments most of the time, we also experienced scenarios where the GA did not converge.

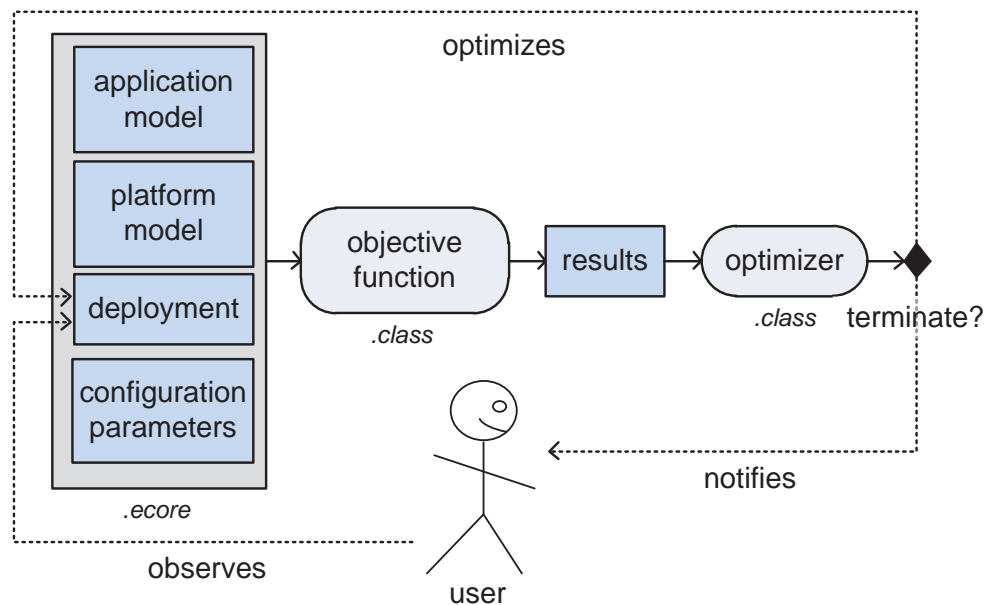


Figure 80: The closed-loop evaluation of the objective function using our second and final prototype.

On a commercially available mobile CPU running with 2.40 GHz, the genetic algorithm terminated on average within 18.5 seconds for the optimization of the above-described real-world example.

7.3 VerSal Implementation

Our implementation of the VerSal technique is based on Eclipse [79, 80] and the Eclipse modeling framework (EMF) [81, 82]. Basically, Eclipse is an extensible platform that is mainly used as a software development environment. However, Eclipse also offers a plug-in development environment to allow the user to develop extensions and integrate them into the Eclipse environment. Our VerSal mediator and the above-

mentioned Eclipse modeling framework were both developed as Eclipse plug-ins.

According to the authors of EMF, *“the eclipse modeling framework is a framework and code generation facility for building tools and other application based on a structured data model.”* To this end, EMF provides a developer with the Ecore meta-model. Using Ecore, the developer is able to specify a custom application model. For such an Ecore-compliant model, EMF offers many automated features, like model serialization, validation, or code generation. We used Ecore for the specification of the VerSal meta-model, which was introduced over the course of chapter 4, and for the specification of the architectural model specified in Appendix A.

Another feature of EMF is its capability of generating a simple but extensible tree-based editor for a user model. We used this feature to develop separate editors for application development and platform development, and for integrating applications and platforms into an open integrated system. The application editor allows an application developer to specify the application’s architectural model together with the application’s VerSal interface. The platform editor offers an analogous functionality to a platform developer. The separation of the two editors allows simulating the separate development of applications and platforms as performed in an open integrated architecture development scenario. The editors use the serialization capability provided by EMF to persist the models using the XML format. The integrator is then capable of importing the XML models of applications and platforms and integrates them using the integrated system editor. Figure 81 shows this process using screenshots of the described editors.

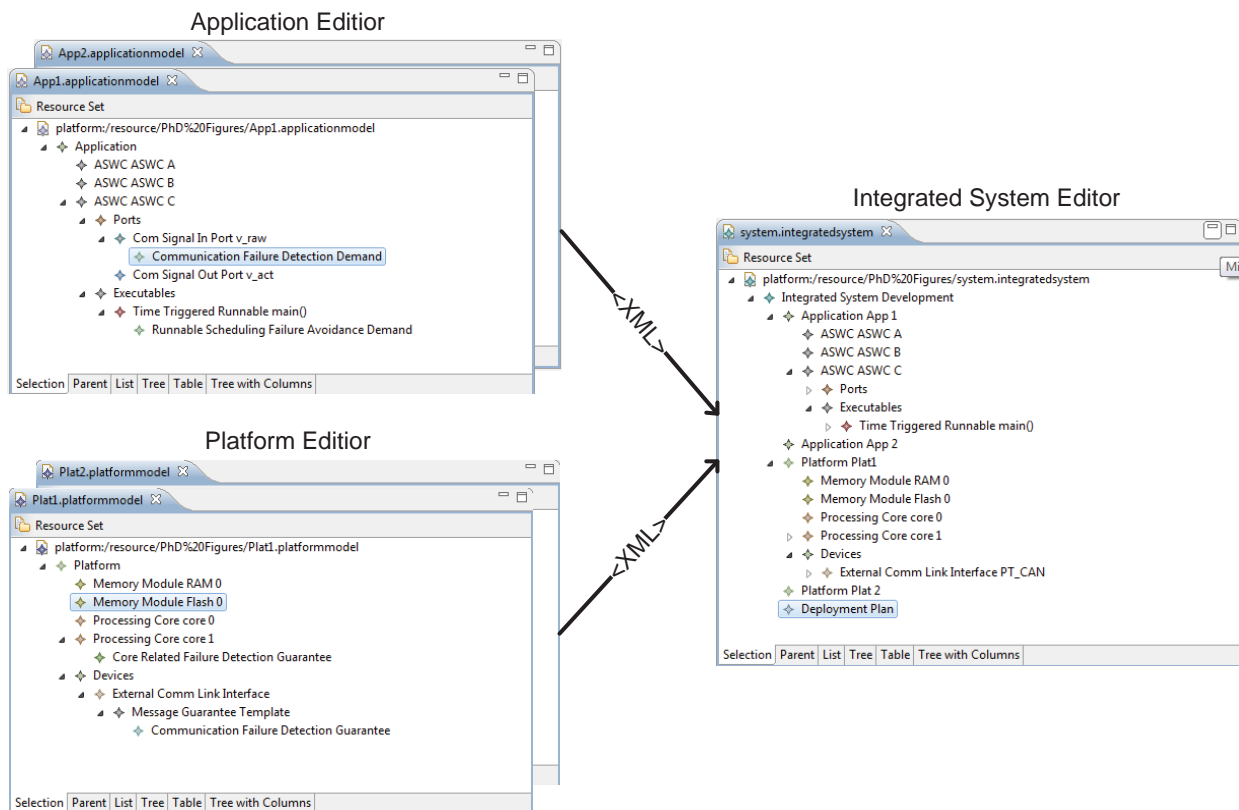


Figure 81: An overview of the VerSal editors provided by the EMF-based implementation of the VerSal technique.

The integrated system editor is the control center from which the integrator controls the mediation. The system editor allows the integrator to configure and integrate the applications and platforms and to toggle the different states of the mediation shown in Figure 62 and described in chapter 5. Whenever the integrator triggers a transition that involves an automatic action of the VerSal mediator, for example the transition between configuration and integration, the system editor automatically executes the appropriate mediator feature, e.g., the automatic evaluation of configuration-dependent conditions. Consequently, the integrator also triggers the actual interface mediation from within the system editor as shown in Figure 82.

After the system editor triggers the interface mediation, the mediator generates a separate mediation report for every application as described in section 5.5.1. The mediation report itself is again based on an Ecore model and is capable of directly referencing the elements of the application and platform models for easy navigation from the report to the model. The implementation of the report viewer is based on a table-tree view provided by the Eclipse platform. Figure 83 shows a screenshot of the mediation report viewer.

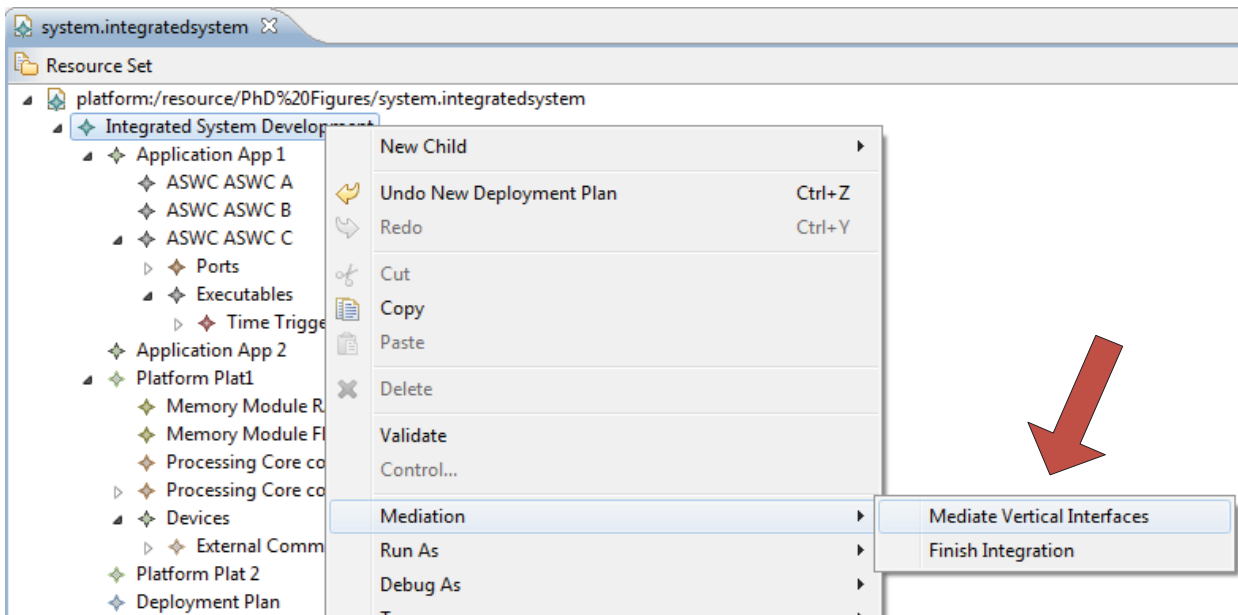


Figure 82: The system editor UI for controlling the mediation

The screenshot shows the 'myApp.mediationResult' window. It displays a table with three columns: 'Demand', 'Fulfillment', and 'Comment'. The table contains a list of demands and their fulfillment status, along with explanatory comments.

Demand	Fulfillment	Comment
myApp	violated	
ProtectionDemand: ASWC_A.signal 1 in	fulfilled	
demand necessity	fulfilled	No condition specified, demand is necessary.
related guarantee existence	fulfilled	1 potentially relevant guarantees available
Protection necessity	fulfilled	The platform contains elements with a lower integrity level, ...
ProtectionGuarantee: message 0x15	fulfilled	
guarantee availability	fulfilled	No condition specified, guarantee is available.
integrity level sufficiency	fulfilled	Guarantee integrity level ASIL_D is sufficient for demand inte...
protection sufficiency	fulfilled	The platform is capable from protecting the application fro...
Check partition level protection	fulfilled	Partitioning is no prerequisite for protection
Check service level protection	fulfilled	Sharing the service with low integrity users violates the prote...
Check element level protection	fulfilled	Sharing the element with low integrity users violates the pro...
ProtectionDemand: ASWC A code section	violated	
ApplicationMonitoringDemand: .ArrivalRateFailure	fulfilled	

Figure 83: The mediation report viewer provided by the EMF-based implementation of the VerSal technique.

7.4 Deployment Implementation

The implementation of our deployment evaluation prototype is based on Eclipse and EMF³³, comparable to the implementation of the VerSal prototype introduced in section 7.2. Among other advantages, sharing the same technology basis will allow us to integrate both prototypes more easily in the future. Comparable to the VerSal meta-model, the deployment evaluation meta-model introduced in section 6.1 Figure 73 was also specified using Ecore.

³³ For more information regarding Eclipse or EMF, please refer to section 7.2.

The tree-based editor automatically generated by the EMF framework provides the user of our prototype with a graphical user interface for specifying deployment problems consisting of a functional architecture and a platform topology. However, in order to find a suitable deployment with the help of a genetic algorithm, we have to translate the specified model-based Ecore representations into a chromosome-based representation. Since we used the Java Genetic Algorithm Package (JGAP) ([83]) to implement the optimization, we had to develop a transformation from the Ecore models into a format predefined by the JGAP developers. After an Ecore model is translated into a chromosome format, JGAP is capable of generating an initial population and performing fitness-based selections and genetic operators like crossover and mutation. Besides some adaptations to our objective function that were necessary to plug it into the JGAP framework, JGAP performed the optimization without further assistance from our side. After the optimization produces a deployment solution, we translate it back into the Ecore format and display it in our EMF-generated editor. The whole process is illustrated in Figure 84.

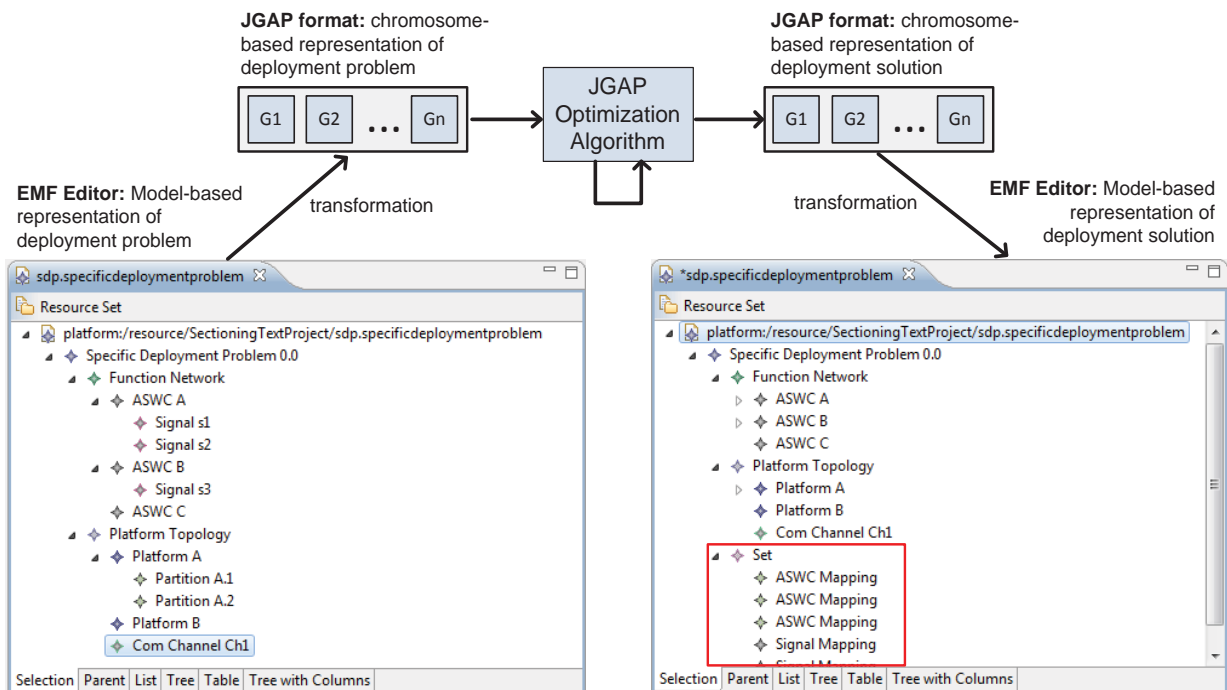


Figure 84: The tool chain used by our deployment optimization prototype

Since the tree-based EMF editor does not provide a good overview for large models, we developed a prototypical graphical user interface using the graphical modeling framework (GMF). Like EMF, GMF is part of the Eclipse modeling project and allows developers to model and generate graphical editors based on EMF and Ecore models. Figure 85 shows our prototypical graphical editor. It illustrates the mapping of ASWCs to partitions using colors. The mapping of signals to communication channels is not depicted by the editor.

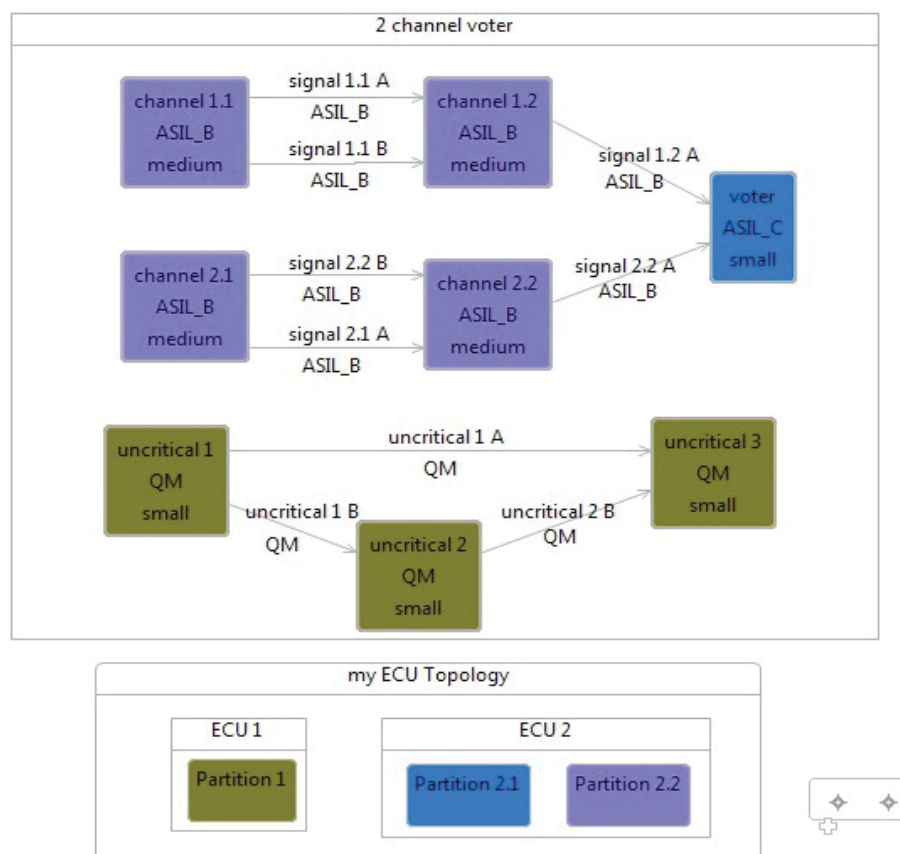


Figure 85: A GMF based visualization of a solved deployment problem (independence of redundant channels has not been considered as a factor for the deployment)

8 Conclusion

In this closing chapter, we will summarize and evaluate the contributions and limitations of our approach (section 8.1), propose possible future areas of work (section 8.2), and conclude this thesis with a final comment (section 8.3).

8.1 Contributions and Limitations

Open integrated architectures like AUTOSAR and IMA allow for more flexible composition of functionality-providing applications and general-purpose execution platforms than traditional federated architectures. Yet if the system is safety-critical, this flexibility is reduced significantly as the safety of the integrated system has to be evaluated whenever the system changes. Therefore, the aim of this thesis was to maintain the flexibility of safety-critical open integrated systems by reducing the safety-related costs when integrating applications and platforms. To this end, this thesis specified and demonstrated a technique for automatically checking the safety compatibility of an application-platform combination as well as an objective function to support the identification of potential application-platform combinations, i.e., deployments.

The technique presented for checking the safety compatibility of application-platform combinations is called *VerSal*, which stands for Vertical Safety Interface. The first of the two parts that constitute our VerSal technique is the *VerSal language*, which allows for formalized model-based specification of the afore-mentioned vertical safety interfaces. In chapter 4, we showed that the VerSal language provides an application-specific component that enables the application developer to modularly specify the application's safety-related demands regarding the behavior of a platform and a complementary platform-specific component allowing the platform developer to specify the safety-related features provided by a platform. Consequently, the VerSal language constitutes our first contribution, which was specified as follows:

Contrib. 1 **Interface Specification:** *Defining a formal language for the modular specification of safety-related demands and guarantees between an application and a platform in an open integrated architecture.*

One main characteristic of the model-based VerSal language is its integration into the design models of applications and platforms. This feature is the key in allowing the second part of the VerSal technique, called the *VerSal mediator*, to automatically reason about the safety

compatibility of an application and a platform. We described in chapter 5 how the VerSal mediator uses the model integration of the VerSal language to navigate from an application demand to an application component, then via the deployment model to a platform resource, and, finally, from this resource via the language integration to the relevant platform guarantees. In the next step, the VerSal mediator checks whether the corresponding demand can be fulfilled using the relevant platform guarantees and displays the result to the system integrator. Overall, the VerSal mediator provides our second contribution, which was specified as:

*Contrib. 2 **Interface Mediation:** Developing an automated process for checking the safety compatibility of an application and a platform in an open integrated architecture.*

As mentioned above, the VerSal mediator requires the planned mapping of application and platform, i.e., the deployment, as input to match demands with their related guarantees. To support the identification of suitable deployment candidates, we introduced a novel objective function for deployment evaluation in chapter 6. This objective function evaluates the aspects of mixed criticality and distribution of safety-critical applications, both inherently tied to the design of an integrated system. The objective function forms our third and final contribution:

*Contrib. 3 **Deployment Evaluation:** Developing an objective function for evaluating and optimizing the deployment of a functional architecture onto a platform topology from a safety perspective.*

While still involving the user, our overall approach automates safety-related aspects of the integration of applications and platforms. However, certain limitations to the approach apply.

In chapter 7, we showed that our language covers the relevant parts of AUTOSAR and ARINC 653. Yet, it is also possible to add application- or domain-specific services to these platforms that are not covered by the public standard. The VerSal language as specified is not capable of covering such services. We addressed this issue by allowing the specification of “free-text” requirements that have to be manually mediated by the user, and by designing the VerSal language and mediator in an extensible way whenever possible.

A second issue is that the VerSal language also does not cover the modular specification of demands and guarantees regarding random failure rates. This is because there already exist various approaches, such as component fault trees [55], that cover this aspect.

As a third limitation, we want to mention that the integration of the VerSal language into the model-based design artifacts of applications and platforms also specifies a requirement for the residual development process. If the product-related design process used does not work with appropriate model-based techniques, the VerSal technique cannot be applied. However, standards like AUTOSAR already prescribe a model-based design methodology, which is why we believe that the market penetration of model-based techniques will further increase.

The fourth and last limitation is the focus of the VerSal approach on vertical safety dependencies, i.e. dependencies between applications and platforms. However, when integrating various applications with each other, we have to check the safety-compatibility on the horizontal level, i.e. between applications as well, which is not covered by VerSal. We think that achieving a comparable level of automation for horizontal safety-dependencies is much more challenging than automatically mediating vertical dependencies as we did. This is because interfaces on functional level are not as standardized as interfaces we find between applications and platforms.

8.2 Future Work

In this section, we will specify two possible future areas of work based on the solutions already provided by this thesis. One addresses the tool-supported integration of deployment optimization and VerSal, the other a problem solver that assists the integrator in case a VerSal mediation fails, i.e., if demands cannot be fulfilled by the given platform.

Currently, deployment evaluation and optimization are decoupled from the VerSal mediation. Deployment optimization identifies potential solution candidates; in the second step, the integrator uses the VerSal technique to check if the given deployment allows for the fulfillment of all safety-related application demands. However, if the mediator fails, there is no automatic feedback for the deployment optimization that filters or reevaluates the incompatible deployment candidates. Here, it would be possible to use the automated VerSal approach for evaluation, transform the currently qualitative answer into a quantitative result, and feed it back as input for the deployment optimizer.

We see three major challenges for this approach: The first is to find a sensible transformation of the qualitative yes/no answers of the mediator to a quantitative result. The second is the scalability of the current mediation algorithm, since the algorithm might be too time-consuming for the evaluation of a large solution space. The third and maybe most difficult challenge lies in bridging the gap between the output of the deployment optimization and the required input of the VerSal mediator. On the one hand, the deployment optimizer produces a mapping of

software components to partitions and signals to busses. The mediator, however, requires a partially configured execution platform. In order to automate this process, a tool would have to automatically configure the platform before being able to use the VerSal mediation.

Such an automated configuration would also help to solve the challenges introduced by the second area of potential future work. In the current system, the mediator produces a result that provides the user with detailed information about why a certain demand is treated as fulfilled or violated. In case a demand is violated, the mediator will also point towards a configuration parameter or a model property that might have “caused” the demand violation. However, the current mediator is not capable of suggesting changes to the configuration, the deployment, or maybe even the implementation of a platform that would help to solve the failed mediation. We think that the development of a component that would help the integrator find solutions for solving a failed mediation would be a challenging yet interesting topic for future research.

8.3 Final Comment

In order to ultimately demonstrate the validity of the VerSal technique, it would need to be applied in an industrial setting. No such direct evidence was produced. Nevertheless, we showed the applicability of the technique through extensive matching of our language to state-of-the-practice standards like AUTOSAR and ARINC 653 and by applying the method using industrial examples provided by our partners. These successful applications and the fact that we automated a process that was performed manually before allow the conclusion that our work is a step towards *“efficiently deploying safety-critical applications onto open-integrated architectures.”*

9 References

- [1] R. Obermaisser, C. El Salloum, B. Huber, and H. Kopetz, "From a Federated to an Integrated Automotive Architecture," *TCAD*, vol. 28, no. 7, pp. 956–965, 2009.
- [2] *ARINC Specification 653 P1-2, Avionic Application Software Standard Interface, Part 1 - Required Services*, ARINC 653 P1-2, 2005.
- [3] AUTOSAR development partnership, *Website of the AUTOSAR Standard*. Available: <http://www.autosar.org/> (2010, Feb. 05).
- [4] H. Fennel, D. K.-P. Schnelle, and P. H. Heinecke, "Achievements and exploitation of the AUTOSAR development partnership," in *Proceedings of the SAE Convergence 2006*: SAE, 2006.
- [5] R. Warrilow, "The avionics platform," Smiths Aerospace, 2004.
- [6] R. Hammett, "Flight-critical distributed systems: design considerations [avionics]," *Aerospace and Electronic Systems Magazine, IEEE*, vol. 18, no. 6, pp. 30–36, 2003.
- [7] *DO-297: Integrated Modular Avionics (IMA) - Development Guidance and Certification Considerations*, RTCA/DO-297, 2005.
- [8] G. Romanski, "Safe and Secure Partitioned Systems and Their Certification," in *Proceedings of the 30th IFAC Workshop on Real-Time Programming and 4th International Workshop on Real-Time Software (WRT/RTS'09)*: IEEE, 2009, pp. 167–172.
- [9] J. Rushby, "Partitioning in Avionics Architectures: Requirements, Mechanisms and Assurance," SRI International - Computer Science Laboratory, 1999.
- [10] US Department of Defense, *Mandatory Procedures for Major Defense Acquisition Programs (MDAPS) and Major Automated Information System (MAIS) Acquisition Programs: DoD 5000.2-R*, 2002.
- [11] Peter Henderson, *Modular Open Systems Architecture*. Available: <http://openpdq.com/MOSAoverview> (2012, Feb. 13).

- [12] A. Sangiovanni-Vincentelli, "Quo Vadis, SLD? Reasoning About the Trends and Challenges of System Level Design," *Proceedings of the IEEE*, vol. 95, no. 3, pp. 467–506, 2007.
- [13] Wind River, "Wind River VxWorks Platforms 6.9 - Product Overview," Feb. 2011.
- [14] QNX Software Systems Limited, *Company website*. Available: <http://www.qnx.com/> (2012, Jun. 03).
- [15] 3S-Smart Software Solutions GmbH, *Company website*. Available: <http://www.3s-software.com> (2013, Jun. 03).
- [16] David N. Kleidermacher, "Real-time Operating System Requirements for Use in Safety Critical Systems," in *EETimes Embedded Systems Conference 2006 (ESC'06)*, 2006.
- [17] SYSGO AG, *Company website*. Available: <http://www.sysgo.com> (2013, Jun. 03).
- [18] *ISO 17356-3: Road vehicles -- Open interface for embedded automotive applications -- Part 3: OSEK/VDX Operating System (OS)*, 2005.
- [19] A. Sangiovanni-Vincentelli and M. Di Natale, "Embedded System Design for Automotive Applications," *IEEE Computer*, vol. 40, no. 10, pp. 42–51, 2007.
- [20] A. Sangiovanni-Vincentelli, L. Carloni, F. de Bernardinis, and M. Sgroi, "Benefits and challenges for platform-based design," in *Proceedings of the 41st annual Design Automation Conference*, New York, NY, USA: ACM, 2004, pp. 409-414.
- [21] L. Carloni, D. de Bernardinis, C. Pinello, A. Sangiovanni-Vincentelli, and M. Sgroi, "Platform-Based Design for Embedded Systems," in *Embedded systems handbook*, R. Zurawski, Ed, Boca Raton: Taylor & Francis, 2006.
- [22] AUTOSAR development partnership, "Virtual Functional Bus (v 2.2.0)," AUTOSAR, Oct. 2011.
- [23] AUTOSAR development partnership, "Layered Software Architecture (v 3.2.0)," Oct. 2011.
- [24] P. Johannessen, "AUTOSAR Safety Approach," in *Proceedings of the SAE Convergence 2006*: SAE, 2006.
- [25] AUTOSAR development partnership, "Technical Safety Concept Status Report (v 1.1.0)," Oct. 2010.

-
- [26] *ARINC Report 651 - "Design Guidance For Integrated Modular Avionics"*, ARINC 651, 1997.
 - [27] *Def Stan 00-74: ASAAC Standards Part 1: Standards for Software*, 2008.
 - [28] P. Binns, "A robust high-performance time partitioning algorithm: the digital engine operating system (DEOS) approach," in *Digital Avionics Systems, 2001. DASC. 20th Conference*, 2001, pp. 1B6/1 -1B6/12 vol.1.
 - [29] A. S. Tanenbaum, *Modern operating systems*, 3rd ed. Upper Saddle River, N.J: Pearson Prentice Hall, 2008.
 - [30] *Common Object Request Broker Architecture (CORBA) Specification - Part 1: CORBA Interfaces*, 2011.
 - [31] *Deployment and Configuration of Component-based Distributed Applications Specification*, 2006.
 - [32] S. Purao, H. Jain, and D. Narareth, "Effective distribution of object-oriented applications," *Communications of the ACM*, vol. 41, pp. 100-108, 1998.
 - [33] C. Pinello, L. Carloni, and A. Sangiovanni-Vincentelli, "Fault-Tolerant Distributed Deployment of Embedded Control Software," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 5, pp. 906–919, 2008.
 - [34] B. Boone, F. de Turck, and B. Dhoedt, "Automated Deployment of Distributed Software Components with Fault Tolerance Guarantees," in *Proc. of the Sixth International Conference on Software Engineering Research, Management and Applications*, 2008. SERA '08, 2008, pp. 21–27.
 - [35] M. C. Bastarrica, R. E. Caballero, S. A. Demurjian, and A. A. Shvartsman, "Two Optimization Techniques for Component-Based Systems Deployment," in *Proceedings of the Thirteenth International Conference on Software Engineering & Knowledge Engineering (SEKE'2001)*, 2001, pp. 153–162.
 - [36] J. A. Bannister and K. S. Trivedi, "Task allocation in fault-tolerant distributed systems," *Acta Informatica*, vol. 20, pp. 261–281, 1983.
 - [37] K. W. Tindell, A. Burns, and A. J. Wellings, "Allocating hard real-time tasks: an NP-hard problem made easy," *Real-Time Syst*, vol. 4, no. 2, pp. 145-165, 1992.

- [38] S. Kartik and C. S. R. Murthy, "Task allocation algorithms for maximizing reliability of distributed computing systems," *IEEE Transactions on Computers*, vol. 46, no. 6, pp. 719–724, 1997.
- [39] S. Shatz and J.-P. Wang, "Models and algorithms for reliability-oriented task-allocation in redundant distributed-computer systems," *Reliability, IEEE Transactions on*, vol. 38, no. 1, pp. 16–27, 1989.
- [40] S. Srinivasan and N. Jha, "Safety and reliability driven task allocation in distributed systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 3, pp. 238–251, 1999.
- [41] S. Kartik and C. Siva Ram Murthy, "Improved task-allocation algorithms to maximize reliability of redundant distributed computing systems," *IEEE Transactions on Reliability*, vol. 44, no. 4, pp. 575–586, 1995.
- [42] *ARP4754A - Guidelines for Development of Civil Aircraft and Systems*, ARP4754A, 2010.
- [43] I. Habli and T. Kelly, "Process and Product Certification Arguments - Getting the Balance Right," *ACM SIGBED Review*, vol. 3, no. 4, pp. 1-8, 2006.
- [44] J. Krodel, "Commercial Off-The-Shelf (COTS) Avionics Software Study," FAA, 2001.
- [45] John M. Rushby, "New challenges in certification for aircraft software," in *Proceedings of the 11th International Conference on Embedded Software, EMSOFT 2011, part of the Seventh Embedded Systems Week, ESWeek 2011, Taipei, Taiwan, October 9-14, 2011*: ACM, 2011, pp. 211–218.
- [46] *Road vehicles - Functional safety*, ISO 26262, 2011.
- [47] *AC 20-148 - Reusable Software Components*, AC 20-148, 2004.
- [48] R. Faller and W. M. Dr. Goble, "Open IEC 61508 Certification of Products," exida GmbH, 2007.
- [49] E. Althammer, E. Schoitsch, G. Sonneck, H. Eriksson, and J. Vinter, "Modular certification support - The DECOS concept of generic safety cases," in *Proceedings of the 6th IEEE International Conference on Industrial Informatics (INDIN'08)*: IEEE, 2008, pp. 258–263.
- [50] T. Kelly and R. Weaver, "The Goal Structuring Notation – A Safety Argument Notation," in *Proceedings of the 34th International Conference on Dependable Systems and Networks (DSN'04)*: IEEE, 2004.

-
- [51] I. Bate and T. Kelly, "Architectural Considerations in the Certification of Modular Systems," in *Proceedings of the 21 st International Conference on Computer Safety, Reliability and Security (SAFECOMP'02)*: Springer, 2002, pp. 303-324.
 - [52] I. Bate, S. Bates, R. Hawkins, T. Kelly, and J. McDermid, "Safety case architectures to complement a contract-based approach to designing safe systems," in *Proceedings of the 21st International System Safety Conference (ISSC'03)*: System Safety Society, 2003, pp. 182–192.
 - [53] W. Damm, A. Metzner, T. Peikenkamp, and A. Votintseva, "Boosting Re-use of Embedded Automotive Applications Through Rich Components," in *Proceedings of the Workshop on Foundations of Interface Technologies 2005 (FIT'05)*, 2005.
 - [54] P. Conmy, "Safety Analysis of Computer Resource Management Software," University of York, York, 2005.
 - [55] D. Domis and M. Trapp, "Integrating Safety Analyses and Component-Based Design," in *Proceedings of the 27th International Conference on Computer Safety, Reliability, and Security (SAFECOMP'08)*: Springer, 2008, pp. 58–71.
 - [56] P. Fenelon, J. A. McDermid, M. Nicolson, and D. J. Pumfrey, "Towards integrated safety analysis and design," *ACM SIGAPP Applied Computing Review - Special issue on safety-critical software*, vol. 2, pp. 21-32, 1994.
 - [57] Y. Papadopoulos, J. McDermid, R. Sasse, and G. Heiner, "Analysis and synthesis of the behaviour of complex programmable electronic systems in conditions of failure," *Elsevier RESS*, no. 71, pp. 229–247, 2001.
 - [58] L. Grunske, "Towards an integration of standard component-based safety evaluation techniques with SaveCCM," in *Proceedings of the 2nd International Conference on Quality of Software Architectures (QoSA'06)*: Springer, 2006, pp. 199–213.
 - [59] *Functional safety of electrical/electronic/programmable electronic safety-related systems*, IEC 61508, 2010.
 - [60] *DO-178C: Software Consideration in Airborne Systems and Equipment Certification*, RTCA/DO-178C, 1993.
 - [61] R. Obermaisser, P. Peti, B. Huber, and C. El Salloum, "DECOS: An Integrated Time-Triggered Architecture," *e&i journal*, vol. 123, no. 3, pp. 83–95, 2006.

- [62] A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis, "Multiple Viewpoint Contract-Based Specification and Design," in *Lecture Notes in Computer Science, Formal Methods for Components and Objects*, F. de Boer, M. Bonsangue, S. Graf, and W.-P. de Roever, Eds.: Springer, 2008, pp. 200–225.
- [63] L. Benvenuti, A. Ferrari, L. Mangeruca, E. Mazzi, R. Passerone, and C. Sofronis, "A Contract-Based Formalism for the Specification of Heterogeneous Systems," in *Proceedings of the Forum on Specification, Verification and Design Languages (FDL'08)*: IEEE, 2008, pp. 142–147.
- [64] J. McDermid and D. Pumfrey, "A Development of Hazard Analysis to aid Software Design," in *Proceedings of the 9th Annual Conference on Computer Assurance (COMPASS '94)*: IEEE, 1994, pp. 17–25.
- [65] J. McDermid and D. Pumfrey, "Assessing the safety of integrity level partitioning in software," in *Proceedings of the 8th Safety-Critical Systems Symposium (SSS'00)*: Springer, 2000, pp. 134–152.
- [66] D. Schneider and M. Trapp, "Conditional safety certificates in open systems," in *Proceedings of the 1st Workshop on Critical Automotive Applications: Robustness & Safety (CARS'10)*: ACM, 2010.
- [67] John M. Rushby, "Runtime Certification," in *Runtime Verification, 8th International Workshop, RV 2008, Budapest, Hungary, March 30, 2008. Selected Papers*: Springer, 2008, pp. 21–35.
- [68] A. Schrijver, *Theory of linear and integer programming*. Chichester, New-York: Wiley, 1998.
- [69] D. E. Goldberg, *Genetic algorithms in search, optimization, and machine learning*: Addison-Wesley, 1989.
- [70] *Specification of Timing Extension (v 1.2.0)*.
- [71] *International vocabulary of metrology -- Basic and general concepts and associated terms (VIM)*, ISO/IEC Guide 99, 2007.
- [72] A. Bondavalli and L. Simoncini, "Failure classification with respect to detection," in *Proceedings of the Second IEEE Workshop on Future Trends of Distributed Computing Systems*, 1990, pp. 47–53.
- [73] *DIN EN 50159 - Railway applications - Communication, signalling and processing systems - Safety-related communication in transmission systems*, 2011.
- [74] *Specification of NVRAM Manager (v 3.2.0)*.

-
- [75] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications (The International Series in Engineering and Computer Science)*: Springer, 1997.
- [76] Certification Authorities Software Team (CAST), "Reliance on Development Assurance Alone When Performing a complex and Full-Time Critical Function: Position Paper CAST-24," FAA, 2006.
- [77] A. Avizienis, J.-C. Laprie, and B. Randell, "Fundamental Concepts of Dependability," LAAS-CNRS, 2001.
- [78] H. Kellermann, G. Németh, J. Kostelezky, K. Barbehön, F. El-Dwaik, and M. Haneberg, "E/E-Architektur der nächsten Generation," *Elektronik Automotive*, vol. Oktober, no. Sonderausgabe BMW 7er, pp. 8–13, 2008.
- [79] E. Gamma and K. Beck, *Contributing to Eclipse: Principles, patterns, and plug-ins*. Boston: Addison-Wesley, 2004.
- [80] Eclipse Project, *Project website*. Available: <http://www.eclipse.org/> (2013, Jun. 03).
- [81] D. Steinberg, *EMF: Eclipse Modeling Framework*, 2nd ed. Upper Saddle River, NJ: Addison-Wesley, 2009.
- [82] Eclipse Modeling Project, *Project website*. Available: <http://www.eclipse.org/modeling/> (2013, Jun. 03).
- [83] JGAP project, *Project website*. Available: <http://jgap.sourceforge.net/> (2013, Jun. 03).
- [84] EAST-ADL Association, "Homepage of the EAST-ADL,"
- [85] *ISO/IEC 14977: Information technology - Syntactic metalanguage - Extended BNF*, ISO/IEC 14977, 1996.

Appendix

Appendix A Architectural Meta-Model

In this Annex we describe our architectural meta-model for the specification of applications and platforms. The architecture model is inspired by existing meta-models like the AUTOSAR meta-model [3] or the EAST-ADL meta-model [84]. Our architecture meta-model is referenced from the VerSal language as shown in Figure 86.

The architecture meta-model consists of four packages that are described separately in the following: First, there is the container architecture meta-model that defines the system-level aspects, which is described in A.1. Second, we introduce the application meta-model that defines application-specific elements in A.2. Third, we sketch the platform meta-model that defines platform-specific elements in A.3. And fourth, we introduce the deployment meta-model that specifies the modeling of the deployment in A.4.

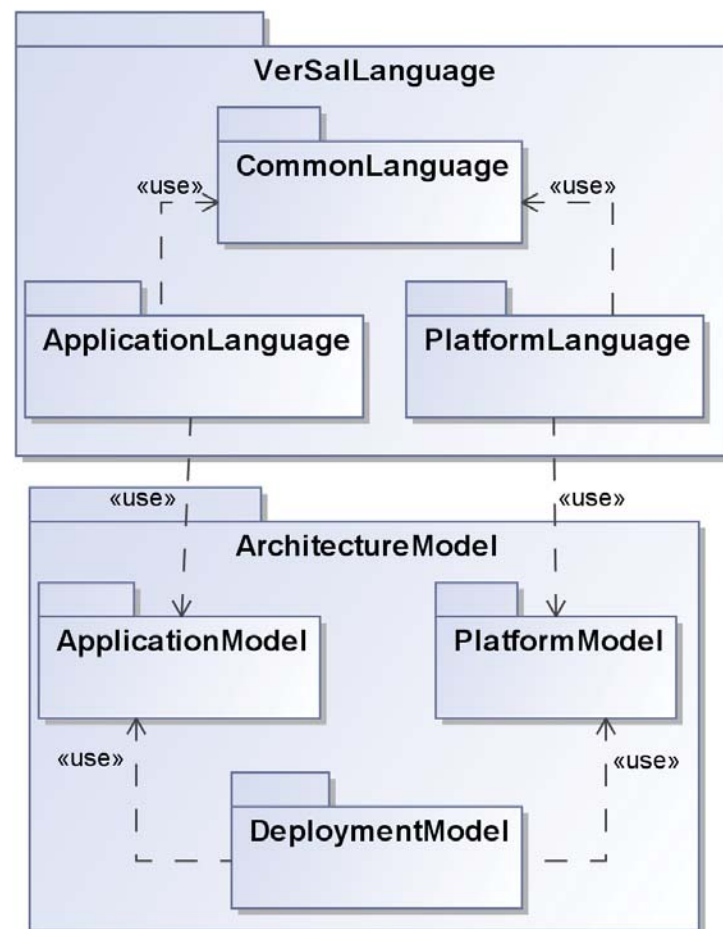


Figure 86: Relations between the VerSal language and the architecture model

In the following we describe the individual classes of the meta-model. In the process we use the following specification conventions: (1) The string "**classA** -> **classB**" notates that the **classA** inherits from **classB**. (2) If a class is abstract it is written in italics "*abstractClass*".

Most of the classes inherit from a class called **NamedElement**, which provides a String attribute called **name**. For reasons of clarity, this repeated inheritance is not denoted in the following.

A.1 System Meta-Model

The system meta-model integrates the platform and application models, as well as the deployment model that links the application with the platform models. It further allows specifying external communication links (i.e. field busses) that connect the different platforms in our system together with the signals that are exchanged between applications. The system meta-model is shown in Figure 87.

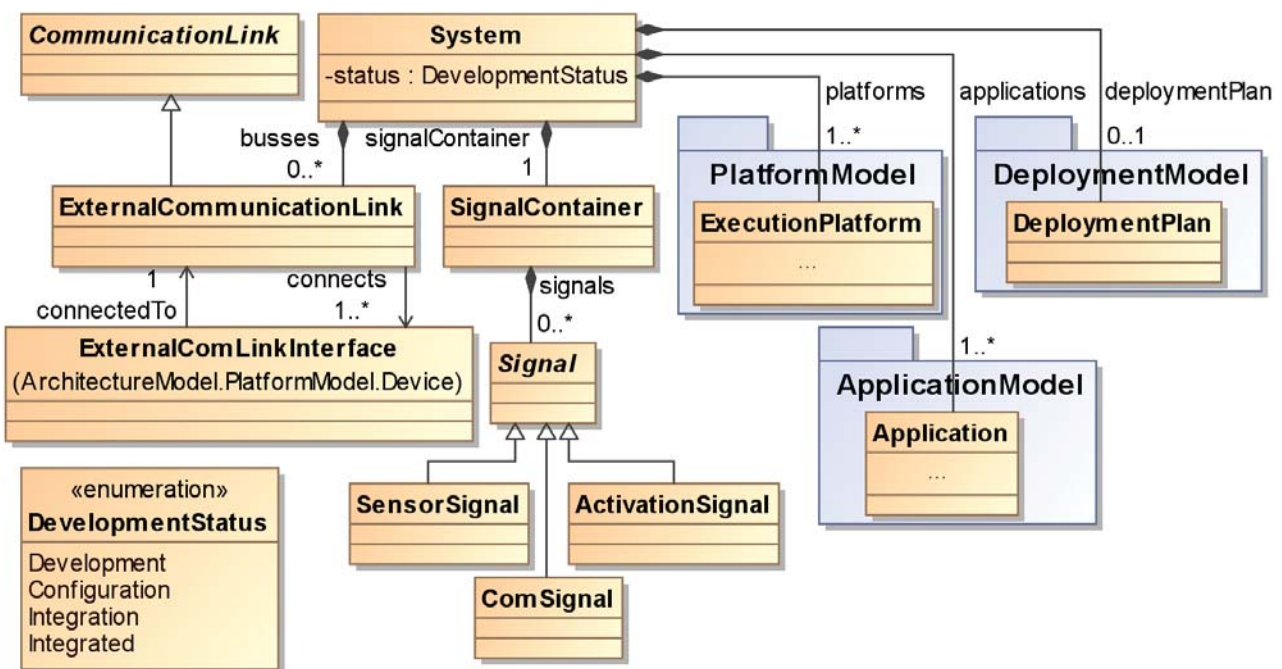


Figure 87: The system meta-model

System

The **System** class is the top-level container class in our meta-model. It contains the execution platforms (see relation **platforms**) and the applications (see relation **applications**) involved in the system, as well as the deployment plan (see relation **deploymentPlan**). It further contains all elements that are not directly assignable to a single platform or application. These are the external communications links (see relation

busses), and the signals that are exchanged between applications (via container class **SignalContainer**, see relation **signalContainer**).

The **System** class further contains a member called **status** that tracks the development status of the system. This status variable controls the different mediation steps as described in chapter 5.

SignalContainer

The class **SignalContainer** is a container for signals (see relation **signals**) that are exchanged between different applications. Signals that are exchanged internally by one application are contained by the corresponding application itself.

Signal

The abstract class **Signal** represents a logical unit of information that is exchanged between components of an application. There are three different types of signals: **SensorSignals**, **ComSignals** and **ActivationSignals**.

SensorSignal -> Signal

A **SensorSignal** is a signal that is produced by a **Sensor** component (e.g. a position sensor, an acceleration sensor ...) and consumed by an **ASWC** (application software component).

ComSignal -> Signal

A **ComSignal** is a signal that is produced by an **ASWC** (application software component) and consumed by an **ASWC**.

ActivationSignal -> Signal

An **ActivationSignal** is a signal that is produced by an **ASWC** (application software component) and consumed by an **Actuator** (e.g. a throttle, a valve ...).

CommunicationLink

A **CommunicationLink** is an abstract class representing an element that is capable of transmitting a **Message**. There are two types of **CommunicationLink**: **ExternalCommunicationLink** and **InternalCommunicationLink** (see platform model A.3).

ExternalComLinkInterface

An **ExternalComLinkInterface** is a special type of **CommunicationLink** that connects different platforms (e.g. a CAN bus). The platforms that are connected by an **ExternalComLinkInterface** are identified via the **connects** relation.

DevelopmentStatus

DevelopmentStatus is an enumeration that lists the different development stages of a **System**, a **Platform** or an **Application**. The different development stages of a system, platform or application are introduced in chapter 5.

A.2 Application Meta-Model

In this section we describe the meta-model used to specify applications. A top-level overview of the application meta-model is shown in Figure 88.

Application

The **Application** class represents an application as defined in RTCA DO-297 [7]: An application is a set of *“software and/or application-specific hardware with a defined set of interfaces that, when integrated with a platform, performs a function”*.

*In accordance with this definition an **Application** contains **Actuators** (see relation **actuators**), **Sensors** (see relation **sensors**) and **ASWCs** (see relation **swComponents**). An application further contains a **SignalContainer** (see relation **signalContainer**) comparable to a **System**. The signal container of an application contains the **Signals** exchanged internally by the respective application.*

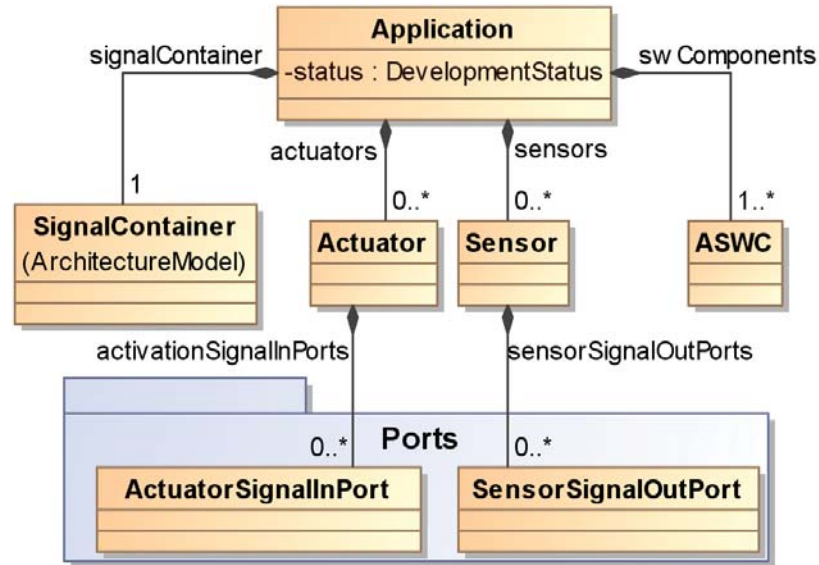


Figure 88: The top-level application meta-model

Actuator

An **Actuator** is a component of an application used to affect its environment (e.g. a motor or a valve). The **Actuator** class contains **ActuatorSignalInPorts** (see relation **activationSignalInPorts**) to model the activation signals expected as input by the actuator component.

Sensor

A **Sensor** is a component of an application used to get information about its environment (e.g. a temperature, pressure or acceleration sensor). The **Sensor** class contains **SensorSignalOutPorts** (see relation **sensorSignalOutPorts**) to model the sensor signals produced as output by the sensor component.

ASWC

An **ASWC** class models a software component of the application containing the ASWC. An ASWC as represented by our meta-model is atomic w.r.t. to its deployment. This means that there can be logical sub-components of an ASWC, but these are of no interest in our case. An overview of the **ASWC** class and its relation is shown in Figure 89.

The interface of the ASWC to other application-level components (i.e. ASWCs, sensors and actuators) is specified by the application's ports. An ASWC has four types of ports: (1) **ActuatorSignalOutPorts** (via reference **activationSignalOutPorts**) to model the interface between the ASWC and actuators. (2) **ComSignalOutPorts** (via

reference **comSignalOutPorts**) to model the signals provided by the ASWC for other ASWCs. (3) **ComSignalInPorts** (via reference **comSignalInPorts**) to model signals consumed by the ASWC provided by other ASWCs. (4) **SensorSignalInPorts** (via reference **sensorSignalInPorts**) to model the interface between the ASWC and sensors.

The **ExecutableEntities** provided by the ASWC are modeled via separate containment references for **Runnables** (see reference **runnables**), and interrupt service routines (**ISR**; see reference **ISRs**).

The direct memory usage of the ASWC is modeled via containment relations of **MemorySections**. We differentiate between memory sections containing code (**CodeSections**; see reference **codeSections**) and memory sections containing data (**DataSection**; see reference **dataSections**). Usually, memory usage is defined directly by an **ExecutableEntity**. However, if code is shared between different **ExecutableEntities** (e.g. a library) or if data are shared between different **ExecutableEntities** (e.g. global variables) this is modeled by a memory section contained by the ASWC.

ServiceNeeds of the ASWC are modeled via the **serviceNeeds** containment reference.

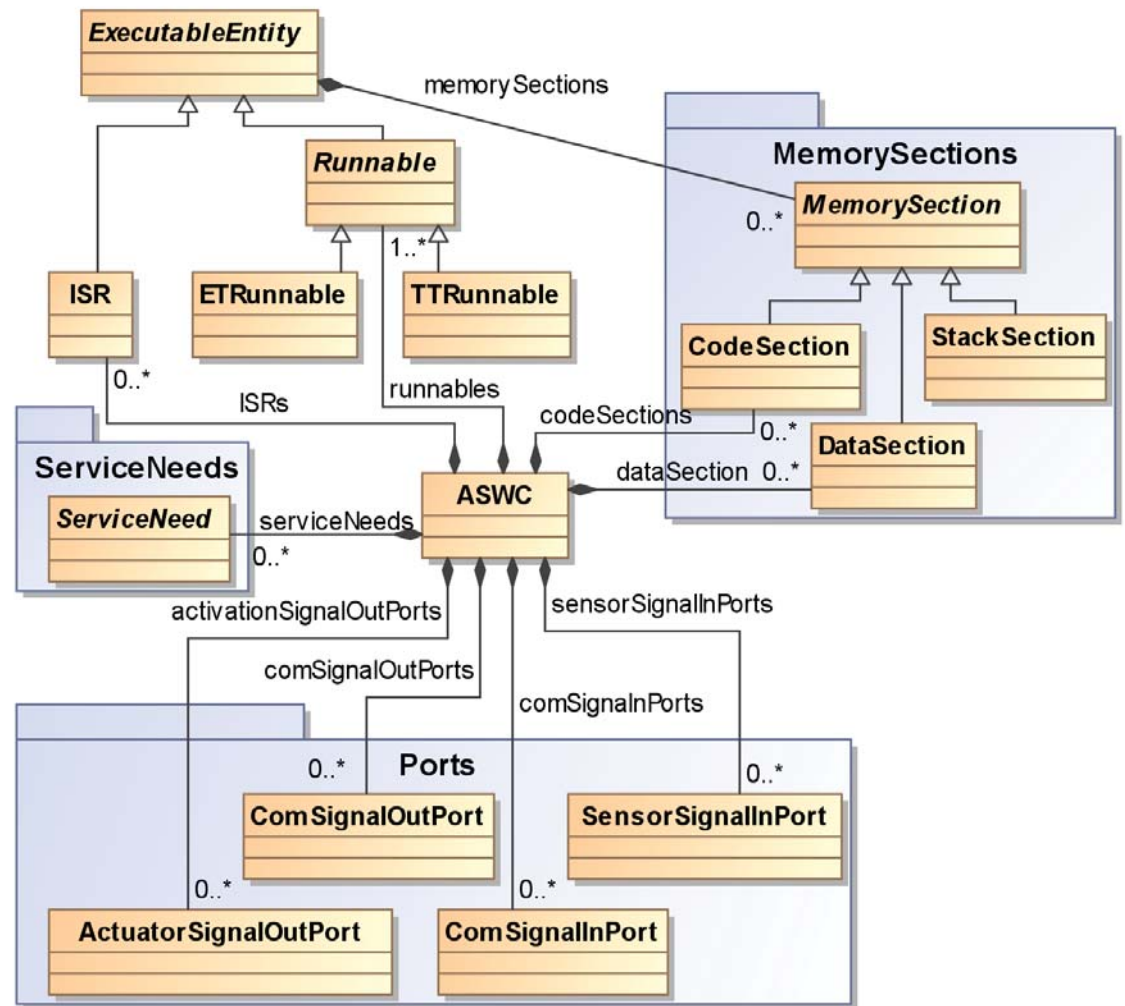


Figure 89: Application meta-model with a focus on the ASWC element

ExecutableEntity

An **ExecutableEntity** represents a function provided by the ASWC that can be individually/concurrently executed by the platform. We differentiate between two kinds of **ExecutableEntities**: **Runnables** and **ISRs**.

The memory usage of an executable entity is modeled by a containment of **MemorySections** (see reference **memorySections**).

ISR -> ExecutableEntity

An **ISR** represents an interrupt service routine, i.e. a software function that can be mapped to a hardware **Interrupt**. This software function is executed when the corresponding interrupt triggers.

Runnable -> ExecutableEntity

A **Runnable** is an abstract class that represents a software function provided by the ASWC that can be individually scheduled by the platform's operating system. We differentiate between two kinds of Runnable: (1) **ETRunnables** and (2) **TTRunables**.

ETRunnable -> Runnable

An **ETRunnable** represents an event-triggered runnable, i.e. a runnable that is activated by an event.

TTRunnable -> Runnable

A **TTRunnable** represents a time-triggered runnable, i.e. a runnable that is periodically activated.

MemorySection

The abstract class **MemorySection** represents a memory section as specified in the object file of the application. We differentiate between three kinds of memory sections: (1) **CodeSection**, (2) **DataSection**, (3) **StackSection** (stack is not defined in the object file but dynamically used and managed by the operating system).

CodeSection -> MemorySection

A **CodeSection** is a memory section containing code (i.e. a set of instructions).

StackSection -> MemorySection

A **StackSection** is a dynamic memory section containing the stack of an executable entity.

DataSection -> MemorySection

A **DataSection** is a memory section containing the data of the application (we do not further differentiate between initialized or uninitialized data. We do not take into account heap memory, since safety-critical systems as regarded by our approach do not allow for dynamic memory allocation).

ServiceNeed

A **ServiceNeed** is an abstract class representing the applications need for a **Service** provided by the platform. In total there are currently eight

different kinds of **ServiceNeeds** specifiable with the VerSal language. An overview is given in Figure 90. Since the different kinds of **ServiceNeeds** are mostly self-explanatory, we will not introduce them separately.

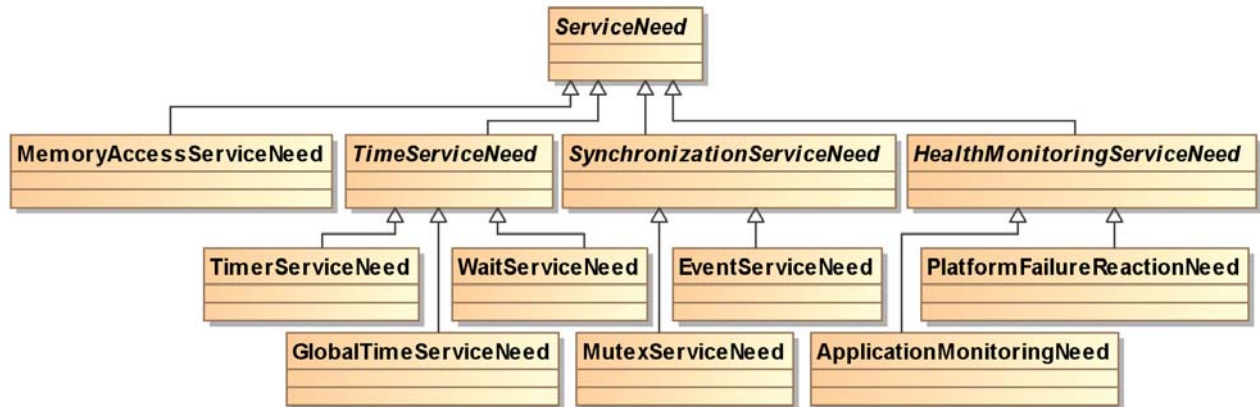


Figure 90: Application meta-model with a focus on the **ServiceNeed** element

Port

A **Port** is an abstract class representing a part of the application's interface to other application elements. Signals are exchanged via ports. We differentiate between **InPorts** and **OutPorts**.

An overview of the meta-model for specifying ports is shown in Figure 91.

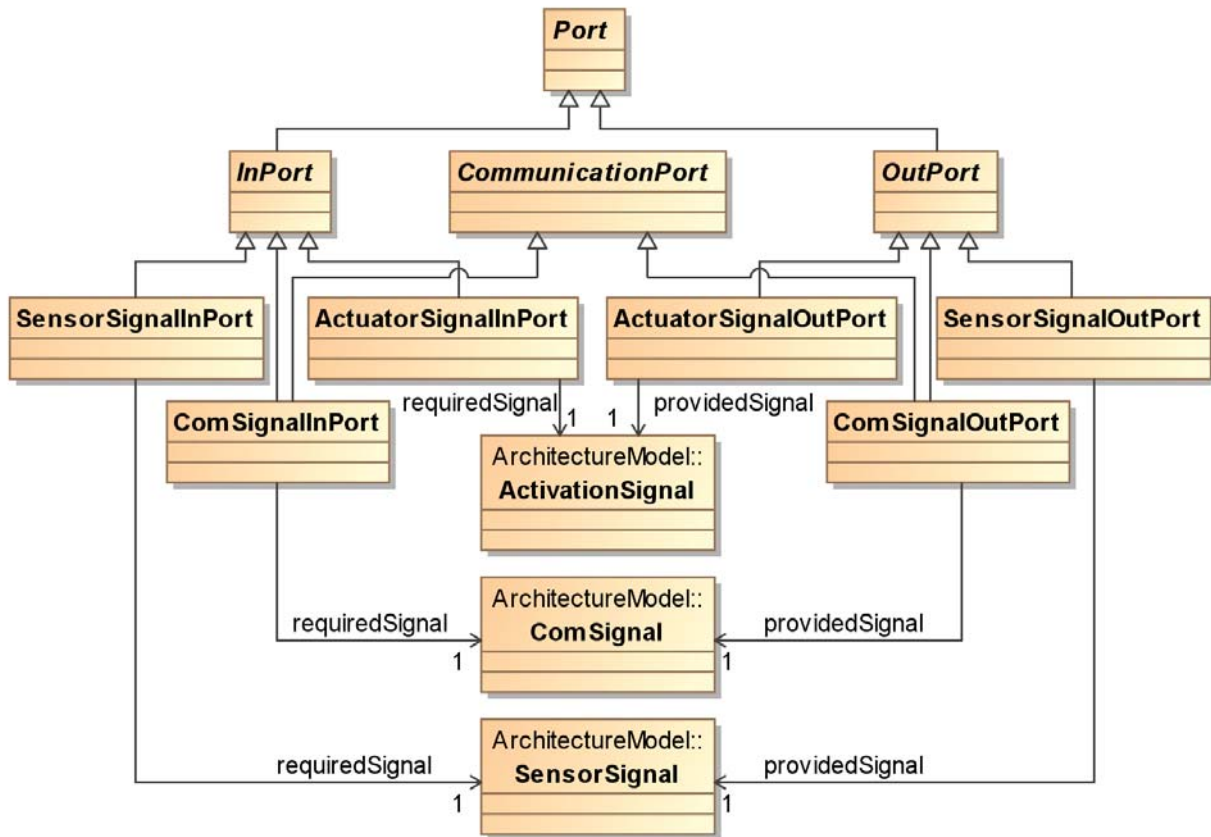


Figure 91: Application meta-model with a focus on the Port element

InPort -> *Port*

An *InPort* is an abstract class representing a piece of information / a signal that is required/consumed by the corresponding ASWC.

OutPort -> *Port*

An *OutPort* is an abstract class representing a piece of information / a signal that is provided/produced by the corresponding ASWC.

CommunicationPort -> *Port*

A *CommunicationPort* is an abstract class specifying that the corresponding port transports **ComSignals**.

SensorSignalInPort -> *InPort*

A *SensorSignalInPort* is a class used to model required/consumed **SensorSignals** (via reference **requiredSignal**) of an ASWC.

SensorSignalOutPort -> OutPort

A **SensorSignalOutPort** is a class used to model provided/produced **SensorSignals** (via reference **providedSignal**) of a **Sensor**.

ComSignalInPort -> InPort, CommunicationPort

A **ComSignalInPort** is a class used to model required/consumed **ComSignals** (via reference **requiredSignal**) of an **ASWC**.

ComSignalOutPort -> OutPort, CommunicationPort

A **ComSignalOutPort** is a class used to model provided/produced **ComSignals** (via reference **providedSignal**) of an **ASWC**.

ActuatorSignalInPort -> InPort

An **ActuatorSignalInPort** is a class used to model required/consumed **ActuatorSignals** (via reference **requiredSignal**) of an **Actuator**.

ActuatorSignalOutPort -> OutPort

An **ActuatorSignalOutPort** is a class used to model provided/produced **ActuatorSignals** (via reference **providedSignal**) of an **ASWC**.

A.3 Platform Meta-Model

In this section we describe the meta-model used to specify platforms. A top-level overview of the application meta-model is shown in Figure 92.

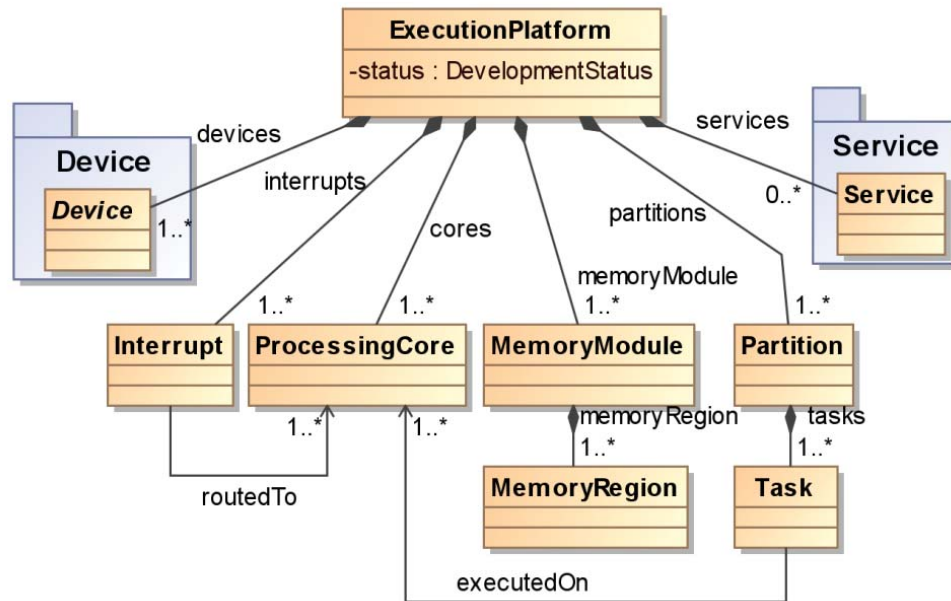


Figure 92: The top-level platform meta-model

Execution Platform

The **ExecutionPlatform** class represents a platform as defined in RTCA DO-297 [7]: A *platform* as a combination of software and hardware to “provide computational, communication, and interface capabilities for hosting at least one application. [...] Platforms by themselves do not provide any [...] functionality”.

An execution platform on the top-level contains **Devices** (via **devices** reference), **Interrupts** (via **interrupts** reference), **ProcessingCores** (via **cores** reference), **MemoryModules** (via reference **memoryModule**), **Partitions** (via **partitions** reference) and **Services** (via **services** reference).

Interrupt

An **Interrupt** is a signal produced by a CPU (synchronously) or by a device (asynchronously), which triggers the execution of an **ISR**.

ProcessingCore

The **ProcessingCore** class represents a CPU/core.

MemoryModule

The **MemoryModule** class represents an individual hardware component providing memory to the system (e.g. a flash, RAM or EEPROM module).

A **MemoryModule** can be logically separated into different **MemoryRegions** (via **memoryRegion** reference).

MemoryRegion

A **MemoryRegion** is a logical compartment of a **MemoryModule**. **MemoryRegions** can, for example, be used to configure a MPU or a MMU to separate and protect different memory regions.

Partition

A **Partition** is a logical compartment that is capable of hosting ASWCs. ASWCs in different Partitions can be protected from each other.

Task

A **Task** is a logical entity that can be individually scheduled by the platform's operating system. **Runnables** of an ASWC can be mapped to **Tasks**.

Device

A **Device** is an abstract class that represents a hardware module that is peripheral to the CPU together with a software stack required to access the device from application level.

Figure 93 gives an overview of the different devices in the platform meta-model.

OutputChannel -> Device

An **OutputChannel** is an abstract class that represents a device together with the software stack needed to physically connect an ASWC to an actuator.

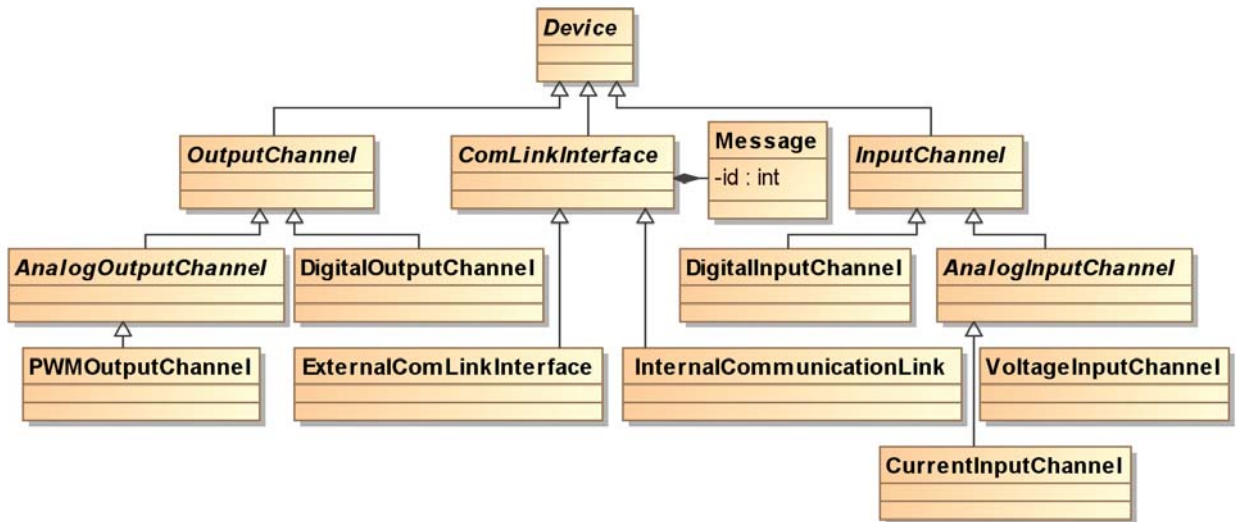


Figure 93: Platform meta-model with a focus on the Device element

AnalogOutputChannel -> OutputChannel

An **AnalogOutputChannel** is an abstract class that represents a device together with the software stack needed to physically connect an ASWC to an actuator driven by an analog signal.

PWMOutputChannel -> AnalogOutputChannel

A **PWMOutputChannel** is a class that represents a device together with the software stack needed to physically connect an ASWC to an actuator driven by a PWM signal.

DigitalOutputChannel -> OutputChannel

A **DigitalOutputChannel** is a class that represents a device together with the software stack needed to physically connect an ASWC to an actuator driven by a digital signal.

ComLinkInterface -> Device

A **ComLinkInterface** is an abstract class that represents a device together with the software stack needed to physically connect an ASWC to a **CommunicationLink**.

A **ComLinkInterface** contains **Message** (see **messages** reference)

Message

A **Message** is an element that transports signals on a communication link. A Message can contain several signals.

ExternalComLinkInterface -> ComLinkInterface

An **ExternalComLinkInterface** is a special type of **ComLinkInterface** that allows connecting a platform to an **ExternalCommunicationLink**.

InternalComLink -> ComLinkInterface -> ComLink

An **InternalComLink** is a special type of **ComLinkInterface** and **ComLink** that allows transporting messages between ASWCs deployed to the same platform.

InputChannel

An **InputChannel** is an abstract class that represents a device together with the software stack needed to physically connect an ASWC to a sensor.

DigitalInputChannel -> InputChannel

A **DigitalInputChannel** is a class that represents a device together with the software stack needed to physically connect an ASWC to a sensor that produces a digital signal.

AnalogInputChannel -> InputChannel

An **AnalogInputChannel** is an abstract class that represents a device together with the software stack needed to physically connect an ASWC to a sensor that produces an analog signal.

CurrentInputChannel -> AnalogInputChannel

A **CurrentInputChannel** is a special type of **AnalogInputChannel** class that represents a device plus software the stack needed to physically connect an ASWC to a sensor that produces a current signal.

VoltageInputChannel -> AnalogInputChannel

A **VoltageInputChannel** is a special type of **AnalogInputChannel** class that represents a device together with the software stack needed to physically connect an ASWC to a sensor that produces a voltage signal.

Service

The **Service** class is an abstract class that represents a mostly software-based functionality provided by an execution platform.

Figure 94 gives an overview of the eight services currently specifiable with the VerSal language. Since the different kinds of **Services** are mostly self-explanatory, we will not introduce them separately.

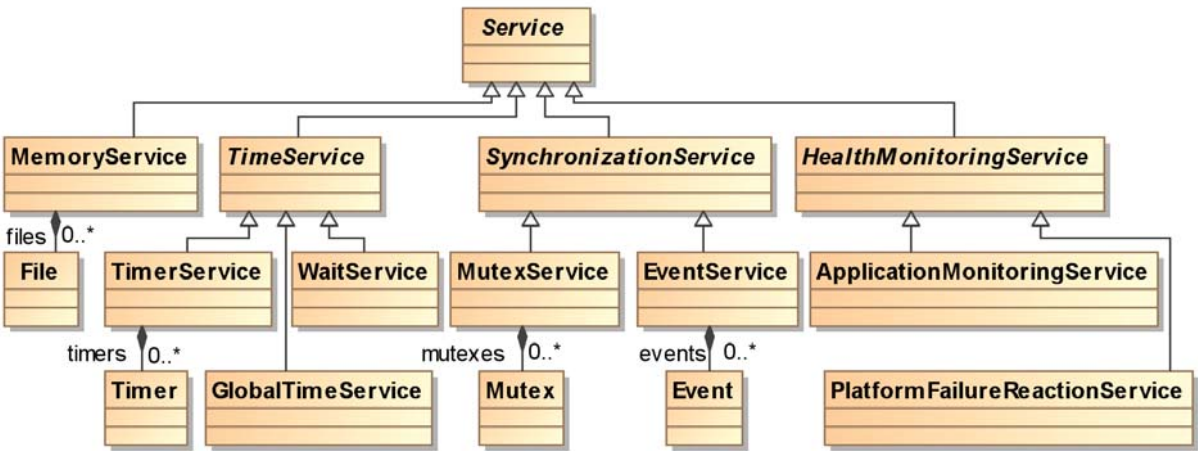


Figure 94: Platform meta-model with a focus on the Service element

A.4 Deployment Meta.Model

In this section we describe the meta-model used to specify a deployment plan. An excerpt of the deployment plan meta-model is shown in Figure 95. Since the model is relatively generic, we only show an excerpt of the model and provide the information w.r.t. the mapping of the different elements in Table 16.

DeploymentPlan

The **DeploymentPlan** class is the central container that holds all mappings between resource users (application elements) and resources (platform elements).

Table 16: All resource mappings specified in the deployment model

<i>Application Element Resource User</i>	<i>Platform Element Resource</i>
ASWC	Partition
MemorySection	MemoryRegion
ExecutableEntity	Task
ISR	Interrupt
CommunicationPort	Message
DigitalSensorInPort	DigitalInputChannel
CurrentSensorInPort	CurrentInputChannel

VoltageSensorInPort	VoltageInputChannel
PWMActuatorOutPort	PWMOutputChannel
DigitalActuatorOutPort	DigitalOutputChannel
TimerServiceNeed	Timer
GlobalTimeServiceNeed	Message
WaitServiceNeed	DigitalInputChannel
MutexServiceNeed	CurrentInputChannel
EventServiceNeed	Event
ApplicationMonitoringNeed	ApplicationMonitoringService
PlatformFailureReaction-Need	PlatformFailureReactionService

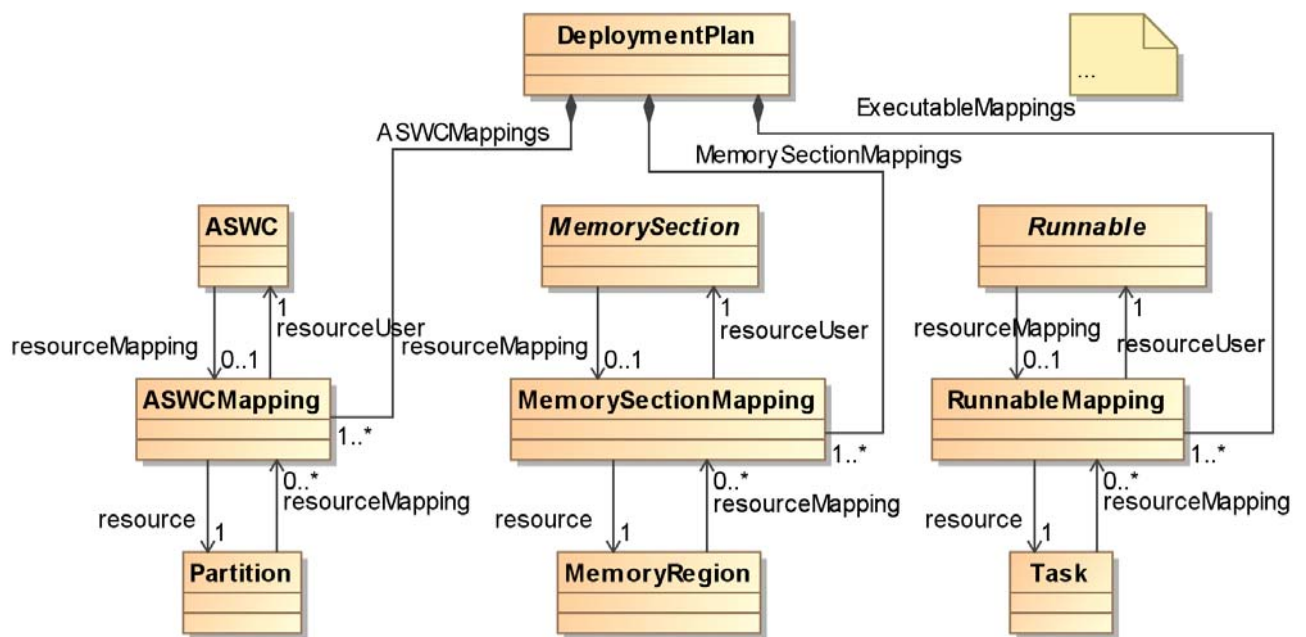


Figure 95: Excerpt of the deployment meta-model

Appendix B List of Common Failure Modes and Reactions

Table 17 lists all failure modes, platform service failure modes as well as application failure modes, that have been specified in sections 4.4.2 and 4.4.3. Table 18 lists the failure reactions specified in section 4.4.4.

Table 17: Failure modes specified in the common language

<i>Service</i>	<i>Id</i>	<i>Name</i>	<i>Failure Class</i>	<i>Parameter</i>
Synchronization	synchFM-1	MutexAccess-Commission	Mutex Failure	
	synchFM-2	MutexAccess-Omission	Mutex Failure	
	synchFM-3	MutexRelease-Commission	Mutex Failure	
	synchFM-4	MutexRelease-Omission	Mutex Failure	
	synchFM-5	MutexTimeout-Failure	Mutex Failure	TimeDeviat
	synchFM-6	EventIndication-Commission	Event Failure	
	synchFM-7	EventIndication-Omission	Event Failure	
	synchFM-8	EventTimeout-Failure	Event Failure	TimeDeviat
Communication	comFM-1	MessageCorruption	Communication Failure	
	comFM-2	MessageInsertion	Communication Failure	
	comFM-3	MessageLoss	Communication Failure	
	comFM-4	IncorrectMessage-Sequence	Communication Failure	
	comFM-5	LateTransmission	Communication Failure	Latency
	comFM-6	EarlyTransmission	Communication	Latency

Input			Failure	
	inFM-1	DigitalInput-Omission	DigitalInput Failure	
	inFM-2	DigitalInputLate-Read	DigitalInput Failure	Latency
	inFM-3	DigitalInputEarly-Read	DigitalInput Failure	Latency
	inFM-4	DigitalInputLate-Return	DigitalInput Failure	Latency
	inFM-5	DigitalInputEarly-Return	DigitalInput Failure	Latency
	inFM-6	DigitalInputFalse-Positive	DigitalInput Failure	
	inFM-7	DigitalInputFalse-Negative	DigitalInput Failure	
	inFM-8	AnalogInput-Omission	AnalogInput Failure	
	inFM-9	AnalogInput-Commission	AnalogInput Failure	
	inFM-10	AnalogInputLate-Sampling	AnalogInput Failure	Latency
	inFM-11	AnalogInputEarly-Sampling	AnalogInput Failure	Latency
	inFM-12	AnalogInput-SamplingJitter	AnalogInput Failure	Jitter
	inFM-13	AnalogInputLate-Return	AnalogInput Failure	Latency
	inFM-14	AnalogInputEarly-Return	AnalogInput Failure	Latency
	inFM-15	AnalogInputValue-Failure	AnalogInput Failure	Error
Output	outFM-1	DigitalOutputLate	DigitalOutput Failure	Latency
	outFM-2	DigitalOutputEarly	DigitalOutput Failure	Latency
	outFM-3	DigitalOutputFalse-Positive	DigitalOutput Failure	

Time	outFM-4	DigitalOutput-FalseNegative	DigitalOutput Failure	
	outFM-5	AnalogOutput-Late	AnalogOutput Failure	Latency
	outFM-6	AnalogOutput-Early	AnalogOutput Failure	Latency
	outFM-7	AnalogOutput-ValueFailure	AnalogOutput Failure	Error
	timeFM-1	GlobalTimeFailure	TimeService Failure	TimeDeviat
	timeFM-2	TimerFailure	TimeService Failure	TimeDeviat
	timeFM-3	WaitTimeFailure	TimeService Failure	TimeDeviat
Memory	memFM-1	MemoryLateRead	MemoryService Failure	Latency
	memFM-2	MemoryRead-AccessDenial	MemoryService Failure	
	memFM-3	MemoryRead-DataFailure	MemoryService Failure	
	memFM-4	MemoryLate-Write	MemoryService Failure	Latency
	memFM-5	MemoryWrite-AccessDenial	MemoryService Failure	
	memFM-6	MemoryWrite-DataFailure	MemoryService Failure	
Scheduling	schedFM-1	SchedulingJitter-Failure	Scheduling Failure	Jitter
	schedFM-2	Scheduling-DeadlineFailure	Scheduling Failure	Latency
	schedFM-3	LateInterrupt-Execution	Scheduling Failure	Latency
Basic	basExFM-1	CPUFailure	CPU Failure	
	basExFM-2	MainMemory-Failure	Main Memory Failure	
	basExFM-3	CPUClockFailure	Clock Failure	
	basExFM-4	PowerSupply-	Platform Failure	

Application Monitoring	Failure			
	appFM-1	ArrivalRateFailure	Application Failure	specific
	appFM-2	InterArivalTime-Failure	Application Failure	Latency
	appFM-3	LogicalSequence-Failure	Application Failure	
	appFM-4	ExecutionTime-Deviation	Application Failure	Latency

Table 18: Failure reaction specified in the common language

Service	Id	Name	Reaction Class	Parameter
Failure Reactions	react-1	Restart Process	Process Reaction	
	react-2	Restart Partition	Partition Reaction	
	react-3	Restart Platform	Platform Reaction	
	react-4	Shutdown Process	Process Reaction	
	react-5	Shutdown Partition	Partition Reaction	
	react-6	Shutdown Platform	Platform Reaction	
	react-7	SendDefault Message	Com-Link Reaction	
	react-8	IssueAnalog Default Signal	AnalogOutput Reaction	specific
	react-9	IssueDigital Default Signal	DigitalOutput Reaction	specific
	react-10	Indicate Failure	ASWC Reaction	
	react-11	Trigger Handler	Executable Reaction	

Appendix C NLR using EBNFs

To enable importing and exporting requirements written in structured text into and from the model-based VerSal language, we developed an example natural language representation (NLR) of the VerSal language. The natural language representation is implemented using the Extended Backus-Naur Form (EBNF).

The principal structure of an EBNF of an interface requirement is as follows:

`uniqueID` = “Text describing the requirement followed by one or many”, **nonTerminalSymbols**, **nonTSymbol** “describing properties and relations”;

`nonTSymbol` = “Production rule of the non-terminal Symbol”;

Each specification of an interface requirement starts with the unique ID of the requirement followed by an equal sign and the definition of the production rule of the requirement. The production rule consists of terminal strings, which are printed in normal face and are encompassed by quotes, and non-terminal symbols which are printed in bold face. The non-terminal symbols are used to specify properties and architectural relations of the requirement. An example of the EBNF of a safety requirement regarding a failure mode of a digital input signal can be found in the following paragraph.

`inFM-1` = “A read omission of the digital input signal”,
 PortName, “must be”, **DemandType**, “.”

Since the naming of non-terminal symbols as well as their production rules align well with the naming and the structure of the meta-model, we will not describe the algorithm for transforming natural language requirements to their model-based representation and back.

The following table gives an overview of the notation of EBNFs that we have used in this document. For further information regarding EBNFs we recommend reading the following standard: [85].

Table 19: The notational elements of the EBNF used in this document

<i>Usage</i>	<i>Notation</i>
definition	=

concatenation	,
termination	;
alternation	
option	[...]
repetition	{ ... }
grouping	(...)
terminal string	" ... "
comment	(* ... *)

Time

In order to specify time-related constraints using the natural language representations of a requirement, we define the following EBNF production rules:

Latency = **CompBound** | **UpperBound** | **LowerBound**;
CompBound = **LowerBound**, "or", **UpperBound**;
UpperBound = "of more than", **Time**;
LowerBound = "of less than", **Time**;
Period = "of more than", **Duration**, "+-", **Jitter**;
IntvalDev = "of more than", **Time**;
Duration = **Time**;
Jitter = **Time**;
Time = **Millisec**, ["", **Microsec**], "ms";
Millisec = digit, {digit};
Microsec = digit, [digit], [digit];
Example: (*a latency*) of more than 9ms or less than 10,999ms
Example: (*a jitter*) of more than 20ms +- 0,5ms
Example: (*a deviation*) of more than 0,5ms

Error

In order to specify error-related constraints using the natural language representations of a requirement, we define the following EBNF production rules:

Error = “larger than”, **AbsoluteErr** | **RelativeErr**;

AbsoluteErr = **AbErrValue**, **Unit**;

RelativeErr = **RelErrValue**, “%”;

AbErrValue = double;

Unit = string;

RelErrValue = integer;

Integrity Level

In order to specify integrity level demands using the natural language representations, we define the following EBNF production rules:

intLevel = “(“, (“QM” | “ASIL A” | “ASIL B” | “ASIL C” | “ASIL D”), “)”;

Platform Service Demands

In the following subsections we will introduce the failure models of the different service classes. Each failure model specification starts with a description of the corresponding service class including a description of the provided functionality and different use-case scenarios of the service class. After the introduction of the service class we illustrate the meta-model that describes the failure model, which includes parameters and architectural references. Before we start describing the individual failure modes of the failure model, we specify the EBNF production rules for the natural language description that are common for all failure modes of the failure model.

Demand = “must be”, **DemandType**, “.”

DemandType = **Detection** | **Avoidance**

Detection = “detected” [“within” **Time**]

Avoidance = “avoided”

Every failure mode has a corresponding EBNF production rule. An exemplary production could look like this:

comFM = “corruption of a message received via port” **PortName**;

The production rule of the failure mode is than always combined with the **Demand** production rule we have specified before, yielding the following production:

comFM-D = “A”, **comFM**, **Demand**;

Example: A corruption of a message received via port *v_act* must be detected within 3ms.

Examples

synchFM-1 = “mutex access commission of”, **SNeedName**;

synchFM-1D = “A”, **synchFM-1**, **Demand**;

Example: A mutex access commission of *mutex_0* must be avoided.

synchFM-2 = “mutex access omission of”, **SNeedName**;

synchFM-2D = “A”, **synchFM-2**, **Demand**;

Example: A mutex access omission of *mutex_0* must be avoided.

comFM-2 = “insertion of a”, **PortType**, “message”, **PortAction**, “via port” **PortName**;

comFM-2D = “An”, **comFM-2**, **Demand**;

Example: An insertion of a required message received via port *esc_available* must be detected.

comFM-5 = “transmission latency of the”, **PortType**, “message”, **PortAction**, “via port” **PortName**;

comFM-5D = “A”, **comFM-5**, **Latency**, **Demand**;

Example: *A transmission latency of the required message received via port esc_Available larger than 2ms must be avoided.*

inFM-7 = A sampling latency of the analog input signal”, **PortName**;

inFM-7D = “A”, inFM-7, Latency, Demand;

Example: *A sampling latency of the input signal throttle_act larger than 0,1ms must be avoided.*

inFM-10 = A value failure of the analog input signal”, **PortName**;

inFM-10D =“A” inFM-10, Error, Demand;

Example: *A value failure of the input signal throttle_act larger than 0,2V must be detected.*

memFM-2 = “denial of a read request entered via”, **SNeedName**;

memFM-2D = “A”, memFM-2, Demand;

Example: *A denial of a read request entered via getErrorCode must be avoided.*

Lebenslauf

Zur Person

Name:	Bastian Zimmer
Anschrift:	Königstraße 91 67655 Kaiserslautern
Geburtsdatum:	06.04.1983
Geburtsort:	Zweibrücken
Familienstand:	Ledig

Arbeit

Seit 10/2008:	Wissenschaftlicher Mitarbeiter am Fraunhofer-Institut für Experimentelles Software Engineering (IESE)
---------------	---

Studium

10/2003 – 09/2008:	Studium der Angewandten Informatik an der Technischen Universität Kaiserslautern Abschluss: Diplom Informatiker (Note: 1,3)
--------------------	---

Zivildienst

01/2003 – 10/2003:	Zivildienst bei der Verbandsgemeinde Schönenberg-Kübelberg
--------------------	---

Schule

08/1995 – 03/2002:	Besuch des Gymnasiums Kusel Abschluss: Allg. Hochschulreife
08/1993 – 06/1995:	Besuch der Realschule Kusel

PhD Theses in Experimental Software Engineering

- Volume 1** **Oliver Laitenberger** (2000), *Cost-Effective Detection of Software Defects Through Perspective-based Inspections*
- Volume 2** **Christian Bunse** (2000), *Pattern-Based Refinement and Translation of Object-Oriented Models to Code*
- Volume 3** **Andreas Birk** (2000), *A Knowledge Management Infrastructure for Systematic Improvement in Software Engineering*
- Volume 4** **Carsten Tautz** (2000), *Customizing Software Engineering Experience Management Systems to Organizational Needs*
- Volume 5** **Erik Kamsties** (2001), *Surfacing Ambiguity in Natural Language Requirements*
- Volume 6** **Christiane Differding** (2001), *Adaptive Measurement Plans for Software Development*
- Volume 7** **Isabella Wieczorek** (2001), *Improved Software Cost Estimation A Robust and Interpretable Modeling Method and a Comprehensive Empirical Investigation*
- Volume 8** **Dietmar Pfahl** (2001), *An Integrated Approach to Simulation-Based Learning in Support of Strategic and Project Management in Software Organisations*
- Volume 9** **Antje von Knethen** (2001), *Change-Oriented Requirements Traceability Support for Evolution of Embedded Systems*
- Volume 10** **Jürgen Münch** (2001), *Muster-basierte Erstellung von Software-Projektplänen*
- Volume 11** **Dirk Muthig** (2002), *A Light-weight Approach Facilitating an Evolutionary Transition Towards Software Product Lines*
- Volume 12** **Klaus Schmid** (2003), *Planning Software Reuse – A Disciplined Scoping Approach for Software Product Lines*
- Volume 13** **Jörg Zettel** (2003), *Anpassbare Methodenassistenz in CASE-Werkzeugen*
- Volume 14** **Ulrike Becker-Kornstaedt** (2004), *Prospect: a Method for Systematic Elicitation of Software Processes*
- Volume 15** **Joachim Bayer** (2004), *View-Based Software Documentation*
- Volume 16** **Markus Nick** (2005), *Experience Maintenance through Closed-Loop Feedback*

- Volume 17** **Jean-François Girard** (2005), *ADORE-AR: Software Architecture Reconstruction with Partitioning and Clustering*
- Volume 18** **Ramin Tavakoli Kolagari** (2006), *Requirements Engineering für Software-Produktlinien eingebetteter, technischer Systeme*
- Volume 19** **Dirk Hamann** (2006), *Towards an Integrated Approach for Software Process Improvement: Combining Software Process Assessment and Software Process Modeling*
- Volume 20** **Bernd Freimut** (2006), *MAGIC: A Hybrid Modeling Approach for Optimizing Inspection Cost-Effectiveness*
- Volume 21** **Mark Müller** (2006), *Analyzing Software Quality Assurance Strategies through Simulation. Development and Empirical Validation of a Simulation Model in an Industrial Software Product Line Organization*
- Volume 22** **Holger Diekmann** (2008), *Software Resource Consumption Engineering for Mass Produced Embedded System Families*
- Volume 23** **Adam Trendowicz** (2008), *Software Effort Estimation with Well-Founded Causal Models*
- Volume 24** **Jens Heidrich** (2008), *Goal-oriented Quantitative Software Project Control*
- Volume 25** **Alexis Ocampo** (2008), *The REMIS Approach to Rationale-based Support for Process Model Evolution*
- Volume 26** **Marcus Trapp** (2008), *Generating User Interfaces for Ambient Intelligence Systems; Introducing Client Types as Adaptation Factor*
- Volume 27** **Christian Denger** (2009), *SafeSpection – A Framework for Systematization and Customization of Software Hazard Identification by Applying Inspection Concepts*
- Volume 28** **Andreas Jedlitschka** (2009), *An Empirical Model of Software Managers' Information Needs for Software Engineering Technology Selection
A Framework to Support Experimentally-based Software Engineering Technology Selection*
- Volume 29** **Eric Ras** (2009), *Learning Spaces: Automatic Context-Aware Enrichment of Software Engineering Experience*
- Volume 30** **Isabel John** (2009), *Pattern-based Documentation Analysis for Software Product Lines*
- Volume 31** **Martín Soto** (2009), *The DeltaProcess Approach to Systematic Software Process Change Management*
- Volume 32** **Ove Armbrust** (2010), *The SCOPE Approach for Scoping Software Processes*

- Volume 33** **Thorsten Keuler** (2010), *An Aspect-Oriented Approach for Improving Architecture Design Efficiency*
- Volume 34** **Jörg Dörr** (2010), *Elicitation of a Complete Set of Non-Functional Requirements*
- Volume 35** **Jens Knodel** (2010), *Sustainable Structures in Software Implementations by Live Compliance Checking*
- Volume 36** **Thomas Patzke** (2011), *Sustainable Evolution of Product Line Infrastructure Code*
- Volume 37** **Ansgar Lamersdorf** (2011), *Model-based Decision Support of Task Allocation in Global Software Development*
- Volume 38** **Ralf Carbon** (2011), *Architecture-Centric Software Producibility Analysis*
- Volume 39** **Florian Schmidt** (2012), *Funktionale Absicherung kamerabasierter Aktiver Fahrerassistenzsysteme durch Hardware-in the-Loop-Tests*
- Volume 40** **Frank Elberzhager** (2012), *A Systematic Integration of Inspection and Testing Processes for Focusing Testing Activities*
- Volume 41** **Matthias Naab** (2012), *Enhancing Architecture Design Methods for Improved Flexibility in Long-Living Information Systems*
- Volume 42** **Marcus Ciolkowski** (2012), *An Approach for Quantitative Aggregation of Evidence from Controlled Experiments in Software Engineering*
- Volume 43** **Igor Menzel** (2012), *Optimizing the Completeness of Textual Requirements Documents in Practice*
- Volume 44** **Sebastian Adam** (2012), *Incorporating Software Product Line Knowledge into Requirements Processes*
- Volume 45** **Kai Höfig** (2012), *Failure-Dependent Timing Analysis – A New Methodology for Probabilistic Worst-Case Execution Time Analysis*
- Volume 46** **Kai Breiner** (2013), *AssistU – A framework for user interaction forensics*
- Volume 47** **Rasmus Adler** (2013), *A model-based approach for exploring the space of adaptation behaviors of safety-related embedded systems*
- Volume 48** **Daniel Schneider** (2014), *Conditional Safety Certification for Open Adaptive Systems*
- Volume 49** **Michail Anastasopoulos** (2013), *Evolution Control for Software Product Lines: An Automation Layer over Configuration Management*
- Volume 50** **Bastian Zimmer** (2014), *Efficiently Deploying Safety-Critical Applications onto Open Integrated Architectures*

Software Engineering has become one of the major foci of Computer Science research in Kaiserslautern, Germany. Both the University of Kaiserslautern's Computer Science Department and the Fraunhofer Institute for Experimental Software Engineering (IESE) conduct research that subscribes to the development of complex software applications based on engineering principles. This requires system and process models for managing complexity, methods and techniques for ensuring product and process quality, and scalable formal methods for modeling and simulating system behavior. To understand the potential and limitations of these technologies, experiments need to be conducted for quantitative and qualitative evaluation and improvement. This line of software engineering research, which is based on the experimental scientific paradigm, is referred to as 'Experimental Software Engineering'.

In this series, we publish PhD theses from the Fraunhofer Institute for Experimental Software Engineering (IESE) and from the Software Engineering Research Groups of the Computer Science Department at the University of Kaiserslautern. PhD theses that originate elsewhere can be included, if accepted by the Editorial Board.

Editor-in-Chief: Prof. Dr. Dieter Rombach

Executive Director of Fraunhofer IESE and Head of the AGSE Group of the Computer Science Department, University of Kaiserslautern

Editorial Board Member: Prof. Dr. Peter Liggesmeyer

Scientific Director of Fraunhofer IESE and Head of the AGDE Group of the Computer Science Department, University of Kaiserslautern

Editorial Board Member: Prof. Dr. Frank Bomarius

Deputy Director of Fraunhofer IESE and Professor for Computer Science at the Department of Engineering, University of Applied Sciences, Kaiserslautern

ISBN 978-3-8396-0753-4

