Automatic Exploration of Automotive E/E Architectures

Johannes Eder

Technischen Universität München
TUM School of Computation, Information and Technology

Automatic Exploration of Automotive E/E Architectures

Johannes D. Eder

Vollständiger Abdruck der von der TUM School of Computation, Information and Technology der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Prof. Dr. Florian Matthes

Prüfer der Dissertation:

1. Prof. Dr. Alexander Pretschner
2. Prof. Dr. Bernhard Rumpe, RWTH Aachen

Die Dissertation wurde am 18.01.2022 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 09.07.2022 angenommen.

Abstract

As the engineering of distributed embedded systems is getting more and more complex, due to increasingly sophisticated functionalities demanding more and more powerful hardware, model-based development of software-intensive embedded systems has become a de-facto standard in recent years. Among other advantages, it enables frontloading techniques like design space exploration which support a system architect already at early stages of development.

In this thesis, we present an approach which is capable of automatically exploring E/E architectures. Based on the concept of viewpoints, we will introduce dedicated technical meta-models, a language to formally describe an E/E architecture exploration problem and an automatic exploration approach using satisfiability modulo theories (SMT). We will furthermore introduce a dedicated methodology and show how such an exploration integrates into a system development process. In the end, we will evaluate our approach by applying it to an industrial use-case provided by Continental.

Acknowledgments

This work would not have been possible without the support of certain people.

My special thanks go to my doctoral supervisor Prof. Dr. Alexander Pretschner for making this PhD possible, for the good discussions and the very valuable and inspiring feedback. But also for the fact that he did not hesitate to take me as a doctoral student from my first supervisor PD Dr. Bernhard Schätz, who unfortunately died too early.

My thanks also go to Prof. Dr. Bernhard Rumpe for very valuable feedback and for taking on the role of second supervisor.

Special thanks also go to Prof. Dr. Sebastian Voss for the great support and continuous feedback over the entire period of the doctorate.

I would also like to thank my colleagues from fortiss and the MbSE competence field, who accompanied me on my way to my doctorate. In particular, Andreas Bayha and Florian Hölzl for the many valuable discussions in our coffee breaks and for your feedback.

Large parts of this work would not have been possible without a deeper insight into industrial reality. My special thanks go to Maged Khalil and Alexandru Ipatiov from Continental for the extremely fruitful collaboration and for providing industrial use cases.

The biggest thanks go to my family, who are always there for me, support me and show understanding even in stressful times.

Last but not least, I want thank PD Dr. Bernhard Schätz, who unfortunately is no longer with us. He gave the decisive impetus for this doctorate.

Munich, January 2022
Johannes Eder

Contents

1. Introduction

The design of distributed, embedded systems is characterized by an ever-growing complexity of those systems. This is caused by an increasing set of functionality, as well as an increase in complexity in the underlying Hardware Architecture and topology. Considering the automotive domain, the introduction of more and more autonomous functions not only increases the amount of software in the vehicles but also demands hardware capable of executing this software.

The so called Electronic/Electric Architecture (E/E Architecture) is a topology of different computational resources, which are referred to as Electronic Control Units (ECUs) and communication resources (buses), which are referred to as buses. The ECUs execute software which is deployed onto them. The buses enable communication between different ECUs. Considering the increasing amount of software due to a rising number especially of autonomous functionality, the trend does not only demand more powerful hardware but also increases the difficulty of integrating such software into hardware. Moreover, there exist multiple variants for each of the *Hardware Resource*s which are differing in their capabilities and most importantly in their costs. Additionally, those E/E Architectures have to comply to development standards like ISO 26262, the safety standard for passenger vehicles. So the design of E/E Architectures for future vehicles gets increasingly complex. In particular, due the fact that the above mentioned aspects of software, hardware and its variants and integration are each getting more complex but still have to be considered jointly.

1.1. Systems engineering and its complexity

In the following, we take a closer look at the rising complexity of systems engineering focusing on the design of E/E Architectures. Thereby, we focus on four different design aspects, namely, the complexity of software and hardware and their integration as well as of variability especially of hardware.

Software complexity The main driver of complexity of software for distributed embedded systems are increasingly sophisticated functionalities, e.g., in the automotive domain, due to the trend towards more and more autonomous vehicles, resulting in a rising number of driver assistance and autonomous functions. In 2007, e.g., a premium car contained already 270 user functions resulting in 2500 "atomic" software functions [1]. A substantial part of this complexity is caused by the intricate dependencies of these functions, complicated by the different, and most often contradicting, requirements of these functions.

Another driver of complexity are new arising use-cases like plug&play capability of systems enabling, for instance, software updates, adding new software and moving software functionality from one ECU to another ECU. As a next step this capability has to be provided over the air. Consequently, further use-cases arise, where vehicle functions are executed in the cloud. Figure 1.1 provides an outline of such future scenarios in the top row.

All of these use-cases share one common problem: it has to be ensured/verified that requirements of the system are still met. This entails that mechanisms have to be provided which enable the verification of those systems at any point in time. As of now these mechanisms can only be provided during development and due to extensive testing.

Hardware complexity The main driver of complexity considering the hardware in distributed embedded systems is the mere number of heterogeneous electrical control units (ECUs) connected via multiple networks (different buses and gateways). In 2007, those automotive E/E Architectures contained up to 67 ECUs [1]. In 2010, they were already consisting of up to 100 ECUs [2][3].

As a result, the trend goes toward more centralized, multi-core architectures as the E/E Architecture would otherwise get too complex, on the one hand, and, on the other hand, increasingly sophisticated driver assistance and autonomous functions require more powerful hardware [4]. Tier-1 suppliers, like Bosch [5], illustrate such a trend (Figure 1.1) from distributed E/E Architectures to domain centralized E/E Architectures and finally to vehicle centralized E/E Architectures. This means that more and more software will be integrated into more and more powerful ECUs, braking the "one function per ECU" paradigm. Regarding the trend depicted by Bosch, the first step consists of domain centralization resulting in domain control units, comprising functionality (software) of certain domains like e.g. powertrain or multimedia. The following step would be vehicle centralization where there are logical control units with a domain independent distribution of functionality (software) enabling adding or moving or removing of certain functionality independent of the control units including functionality which can be executed in the cloud. In the aerospace domain, e.g., the concept of centralized architectures can already be found in the IMA [6] or ARINC 653 [7] standard.

Consequently, E/E Architectures are changing from approximately 100 ECUs connected via various buses and gateways to smaller more centralized architectures with more powerful resources considering e.g. computational capabilities. Yet, a figure like 1.1 only proposes a vague vision of an E/E architecture of the future. In particular, how the different steps will be accomplished from Modular, Integration, Domain Centralization, Domain Fusion, Vehicle Fusion and Vehicle Computing is thus an open question. Thus, the design of such architectures as an interplay of software and hardware gets increasingly complex.

Integration complexity The development according to a standard like ISO 26262 as mentioned before, distinguishes between software and hardware. Hence the integration of both

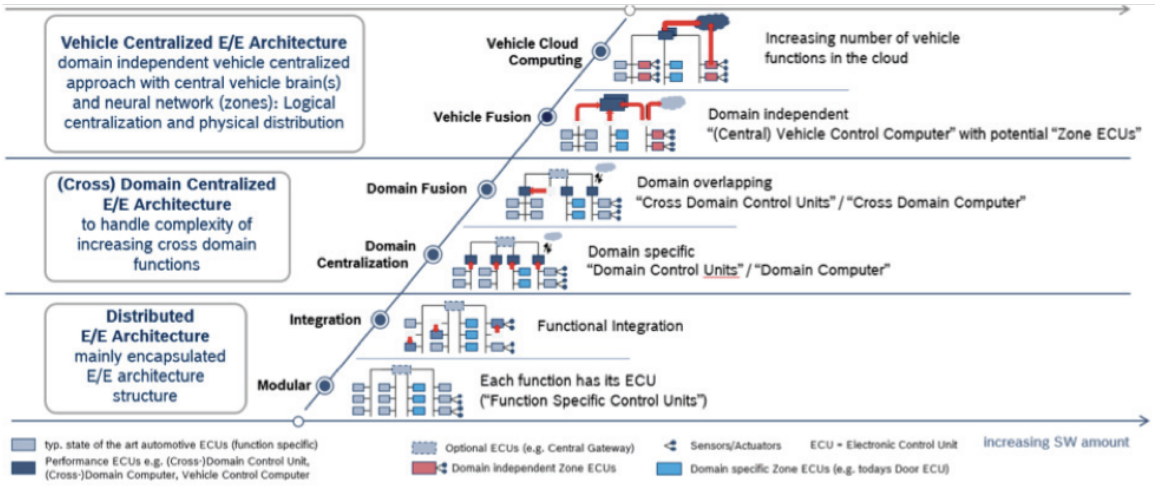


Figure 1.1.: Trends in Automotive E/E Architectures [5]

is a substantial part. One part of the this integration is the deployment of software to hardware where the software will be executed. Considering a distributed embedded system this deployment has a big implication as it poses a certain demand on the communication resources which are used, due to the fact, that executing a certain software function on one ECU in the system may require input from another software function located on a different ECU. Moreover, such deployment has to meet given requirements (e.g. busload, safety, etc.) which have to hold. Recapitulating, that the number of software functions is ever growing while the E/E Architecture of the future will change from large distributed to more centralized systems, the integration of both software and hardware - and consequently the overall system design - is getting more complex, too.

Variability complexity Variability is a fourth driver of complexity as products are not only more and more tailored towards individual needs but also due to regionalization, covering aspects such as customer needs and legal issues.

Considering (software) functions of a system, the number functions differs due to customer needs and region specific variants. This means that some functions may be optional and some functions may be required only in specific regions or even forbidden in some other regions.

Considering the system hardware, the the set of deployed functions may have an impact as e.g. less functions may require less ECUs resulting in a cheaper system/product. On the other hand, parts of the hardware (e.g. ECUs and buses) may exist in different variants considering cost, performance and safety attributes to mention just a few.

Thus during system design, one has to cope with the rising demand of individualization of products, while at the same time trying to minimize, in particular, the cost of the product

regarding all possible product variants.

1.2. Problem statement

In order to effectively manage the ever increasing complexity in the design of embedded systems, development processes in general, and model-based approaches in particular, support the development. They are assuming an idealized (component-based) model of computation, abstracting away from implementation issues like interference aspects of the execution platform resulting from shared computation or memory resources.

One of the biggest advantage of such model-based approaches - in contrast to document based approaches - is the possibility to provide formalisms to enable the automation of development steps. However, as system models provide an abstraction and are thus simplifying the physical system, ISO 26262, e.g., demands that the assumptions behind those abstractions are not violated which has to be ensured in each development step. For instance, during SW-/HW-integration, platform mechanisms must avoid an overload in bus communication, such that messages do not arrive too late or even get lost. The distribution of software across different execution platforms is thus highly affecting different aspects of such an integration. Due to the rising complexity of today's E/E Architectures caused by more and more complex functions that have to integrated into increasingly powerful hardware and the trend towards more centralized E/E Architectures, there is a need for automated support during system design.In this thesis, we therefore answer one of the major industrial needs for an automated system design support by answering the following overall question:

"How can we support a system architect during the design of future E/E Architectures?"

The definition of "how" certainly involves an automated design support regarding the rising complexity of today's E/E Architectures which can no longer be dealt with manually. Moreover, the given system complexity influences the development in a way that development approaches/processes have to be capable of appropriately supporting the design of E/E Architectures which also demands sound methodologies. Hence, it is not enough to just provide an automation for a certain development step, but also a methodology telling how an automatized support works and which can be integrated into a development approach/process. As model-based approaches provide formalism which enable automation, we need dedicated models which precisely define an E/E Architecture. This entails that the following research question has to be answered:

RQ1 Which **models** are needed to precisely describe an E/E Architecture and how are they defined?

A model-based definition is the first step towards a semi-automatic design of E/E Architectures. It enables a formal definition of both the architecture and requirements for the design of the architectures that need to be satisfied. Consequently, this leads to a second research question, which as to be answered:

RQ2 How to **formalize** E/E Architecture models and how to formalize requirements which have to be satisfied by an E/E Architecture?

A formalization of models and their requirements is needed, in order to enable an exploration of E/E Architecture, meaning a semi-automatic approach to automatically synthesize E/E Architectures. exploration problem which can then be solved automatically. Furthermore, a detailed understanding of today's E/E Architectures is required.

First of all, in order to find an optimized or even optimal E/E Architecture, variability aspects have to be considered. This entails taking into account all possible variants of *Hardware Resource*s, namely computation or communication resources. For example, a computation resource like a sensor or actuator may exist in different variants. Those variants could be differing in properties such as cost or performance, but also in the set of possible interfaces enabling connections to differing communication resources (e.g. CAN buses). A decision in favor of a certain variant is thus affecting the whole E/E Architecture as its interface limits the set of possible communication resources that can be used.

Second of all, striving for an optimal E/E Architecture is furthermore closely intertwined with the deployment of software to computation resources, due to the fact that, software is demanding a certain amount of computational resources (memory, performance,..) in order to be executed. In the same way, communication resources (e.g. buses) provide a certain throughput of information which is required for exchanging messages between the software of the different computational resources.

By targeting the industrial need of reducing the complexity in systems engineering and especially E/E Architecture design, more automation needs to be introduced in the engineering processes. This requires exploration approaches that, based on well defined models, enable an automatic exploration of the deployment of software tasks to computation resources and of software signals to communication resources. At the same time, variability aspects of *Hardware Resource*s have to be considered as well, which demands answering the following research question:

RQ3 How can we formally define an E/E Architecture **exploration** problem which enables the calculation of valid and optimized or even optimal E/E Architectures taking into account deployment and variability aspects?

The capability of modeling, formalizing and automatic exploration enables to considerably support a system architect in developing an E/E Architecture. However, those three aspects have to be connected to provide a sound and reproducible approach. Hence, it has to be

ensured that those aspects can be integrated into an existing development approach. Thus, we have to finally provide an answer to the following question:

RQ4 How does a dedicated exploration **methodology** for E/E Architectures look like and how does it integrate into an existing development process?

Based on answering these four research questions, we describe the contribution of this thesis in the following.

1.3. Contribution

Based on the provided research questions, the contribution of this thesis emerges in a gap to the state-of-the-art. Section 1.3.1 will give an overview of this gap. (A detailed overview of the related work of this thesis will be given in Chapter 3.) In Section 1.3.2, we describe the contribution of this thesis in terms of the identified gap to SOTA relating them to the four research questions.

1.3.1. Gap to SOTA

Architecture description languages This thesis provides a E/E Architecture exploration, thus the architecture description languages (ADLs) are a core principle. Considering such ADLs, there are a few wide-spread languages in particular the Unified Modeling Language UML [8]. There are two main derivatives of this language: the "Systems Modeling Language" *SysML* [9], an extension focusing on system modeling and "Modeling and Analysis of Real-Time and Embedded Systems" *Marte* [10] an extension focusing on real time and scheduling aspects. The UML is a general purpose modeling language which does not include any hardware specific aspects and mainly targets software modeling. The SysML has been built to enable systems modeling. However, it also does not provide any E/E specific hardware related models which enable modeling of E/E Architectures. UML Marte distinguishes between different hardware elements (processor, bus) and provides deployment models. However, all three languages do not support variant modeling and thus do not support modeling variability aspects.

The" AUTomotive Open System ARchitecture" AUTOSAR [11], is a modeling language which has been explicitly created for the automotive domain. Although this modeling approach especially targets E/E Architectures, it focuses on implementation specific aspects which makes it unsuitable to use in early stages of development (frontloading) and furthermore provides no variant modeling capability. The East ADL2 [12] provides a high level abstraction of AUTOSAR enables modeling a E/E Architecture and also variability aspects. However, the language is missing the consideration of the software when modeling an E/E Architecture.

Lastly, the "Architecture Analysis & Design Language" AADL [13] is a modeling language which has been built for the aerospace domain and supports modeling Hardware

Architectures and Software Architectures and also deployment aspects. However, it also does not support variant modeling.

This means that there is a lack of adequate architecture description languages targeting automotive E/E Architectures. Section 3.1 gives a more detailed overview over these architecture description languages.

Domain specific languages Regarding the area of languages which enable the formalization of models, there is mainly the Objective Constraint Language (OCL) [14] enabling the formalization of constraints for UML Models. However this language does not provide formalisms for optimization and is a comprehensive standard which does not capture any domain specificities of automotive E/E Architectures.

On the other hand, a language like SMTlib2 (Satisfiability Modulo Theories Library) [15] provides a formalism which is bound the use of SMT solvers and is not intended to cover automotive domain specific aspects.

This entails that there is no domain specific language which would enable a formalized description of E/E Architectures. Section 3.2 gives a more detailed overview over those languages.

Exploration/synthesis approaches A variety of work has been conducted in the area of design space exploration. Especially the deployment problem deploying software units (components, tasks, etc.), representing a certain functionality, on computation units has been widely studied. Although some of them are variability aspects of the deployment, a lot of them are considering a fixed *Hardware Topology*.

There is also variety of work on the topology of Hardware Architectures which is close to the exploration of automotive E/E Architectures. However, those techniques either do not consider the deployment aspects which can heavily influence the creation of an E/E Architecture or they are not taking into account different variants of hardware parts. Exploring *Hardware Topologies* has been studied mainly on a system-on-chip level and not on a distributed system level like the automotive E/E Architecture, focusing only on the specifics of one hardware part.

Lastly, there are also generic exploration approaches which, however, do not cover domain specific aspects and are thus not usable for exploring E/E Architectures.

This thesis focuses on a combined exploration approach which solves E/E Architecture exploration problem. Thus we are taking into account an appropriate architecture description language, a domain specific language in order to formalize the problem and a dedicated exploration approach which can automatically calculate E/E Architectures. Section 3.3 gives a detailed overview over design space exploration approaches.

Engineering methodologies There a few model-based engineering methodologies, which propose approaches in order to develop systems in a model-based way. SPES [16] and CESAR [17] are two academic approaches which propose system modeling on different levels

of abstraction. IBM Harmony [18] and the AUTOSAR methodology [19] are industrial methodologies which propose specifying modeling approaches for the SysML and the AUTOSAR architecture modeling language. However, they are not providing a methodology enabling exploration in order to automatically calculate (E/E) architectures.

To the best of our knowledge, there are no system development methodologies which actively enable the use of exploration techniques. Section 3.4 gives a more detailed overview over the four mentioned methodologies.

In summary, a comprehensive automated exploration approach, considering modeling, language, exploration as well as methodological aspects, is yet missing.

1.3.2. Contribution

The contribution of this thesis is the provision of a holistic exploration approach for E/E Architectures, answering the overall problem statement (Section 1.2) of how a system architect can be supported during the development of future E/E Architectures. This entails the definition of dedicated models for modeling the E/E Architecture. Furthermore, a dedicated domain specific language which is capable of formalizing those models and which is enabling the automatic calculation of E/E Architectures. Thirdly, an exploration which is capable of processing this formalized description in order to automatically calculate optimal E/E Architectures. And lastly a methodology, which provides an integration of models language and exploration and integrates into an existing development approach.

Specifically, this thesis provides the following contributions.

1. A E/E Architecture Viewpoint which provides meta-models for modeling an E/E Architecture on different levels of abstraction: software, hardware and deployment of software to hardware. The software model provide a meta-models for Tasks and Signals which together form a Task Architecture. The hardware models provide meta-models for modeling the hardware of a E/E Architecture in terms of computation and communication resources and their variants and meta-models for modeling the topology of an E/E Architecture. (RQ1)

2. A Specification Viewpoint which provides a domain specific language which is able to formally describe the models defined the E/E Architecture Viewpoint to description exploration problem. Moreover, this specification is able to define optimization objectives which enable an optimization of E/E Architectures and constraints which enable the formal description of requirements which have to hold when modeling an E/E Architecture. (RQ2)

3. An Exploration Viewpoint which provides a translation of this language into SMT in order to be solved by the state of the art SMT solver Z3 developed at Microsoft. This viewpoint furthermore enables to find contradicting constraints (validation) and to calculate optimal E/E Architectures in terms of the topology and the deployment of software to hardware (exploration). (RQ3)

4. A methodology which provides a holistic architecture exploration approach, on the one hand integrating the three viewpoint to form a sound exploration approach, and, on the other hand, integrating this exploration approach into a standard development process and into a existing systems engineering methodology. (RQ4)

1.4. Solution

Based on these four contributions, the intended solution of this thesis is schematically shown in figure 1.2.

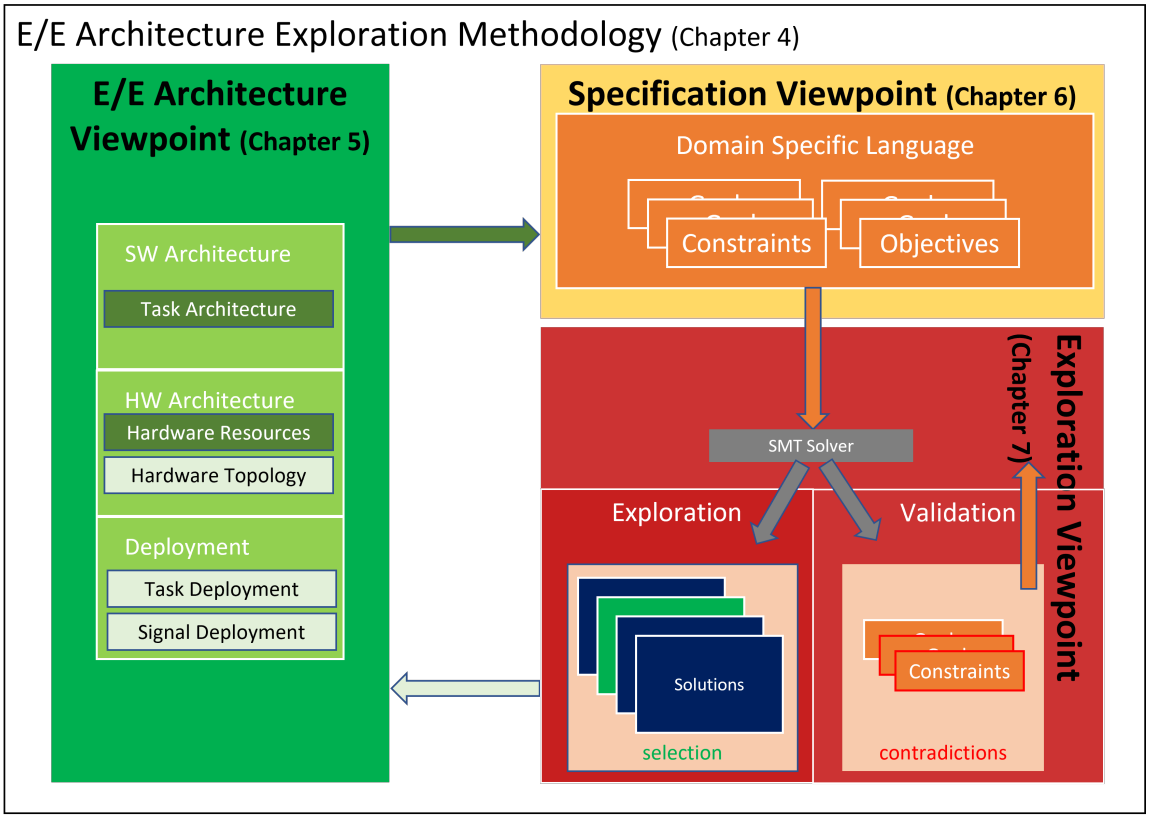


Figure 1.2.: Schematic illustration of the solution and structure of this thesis

The E/E Architecture Viewpoint, consists of different models for software, hardware and deployment - enabling a comprehensive hardware description - is the basis for the definition of E/E Architecture models. Specifically, the task architecture as well as the *Hardware Resources* (ECUs, Buses and variants of both) serve as input for the Specification Viewpoint.

The exploration problem is formalized in the Specification Viewpoint using a domain specific language (DSL). This DSL enables the formalization of the architecture models in

terms of constraints, which restrict the set of possible architectures and objectives, enabling optimization of architectures. Additionally, requirements for an E/E Architecture can be formalized in this language as constraints and objectives.

The Exploration Viewpoint, transforms the formalized description into a solving techniques which is capable of automatically calculating the constraints, finding contradictions between them and calculating optimal architecture solutions. Due to the fact that the Specification Viewpoint is situated between E/E Architecture and Exploration Viewpoint, we divide problem description and solution calculation which enables to use any possible solving technique. In this thesis, we will use the Z3 SMT solver. However, e.g. Integer Linear Programming (ILP) or evolutionary algorithm approaches could be used as well. The solutions during the exploration calculate optimal *Hardware Topologies* and the related Task and Signal Deployments onto the *Hardware Topologies* which corresponds to an E/E Architecture.

As briefly described above, the methodology describes the interaction between the different viewpoints and how they can be integrated into an existing development methodology SPES and into a customized V-Model XT development process.

1.5. Publications

This thesis is based on the following publications.

- MODELS Conference 2016, OSS4MDE workshop:
  Eder et al. "Usable Design Space Exploration in AutoFOCUS3" [20]
- MODELS Conference 2017, practice and innovation track:
  Eder et al. "Bringing DSE to life: exploring the design space of an industrial automotive use-case" [21]
- MODELS Conference 2018, doctoral symposium:
  Eder et al. "Exploration of hardware topologies based on functions, variability and timing" [22]
- MODELS Conference 2018, practice and innovation track:
  Eder et al. "From deployment to platform synthesis - automatic synthesis of distributed automotive architectures" [23]
- SoSyM Journal 2020, special issue:
  Eder et al. "Hardware Architecture Exploration - Automatic Exploration of Distributed Automotive Hardware Architectures" [24]

1.6. Outline

In the following, we depict the outline of this thesis. Figure 1.2 serve as reference, as it shows the distribution of the contribution over Chapters 4 - 7.

Chapter 1 introduces this thesis by providing a motivation, problem statement, contribution and solution which summarizes the following chapters.

In Chapter 2 we are introducing relevant aspects when developing automotive E/E Architectures. This entails especially how those architectures are developed in compliance with ISO26262 which is the safety norm for passenger vehicles, how they are developed in a model-based way and how the current trends of E/E Architectures predict.

Chapter 3 describes the related work of thesis which is based on the gap to the state of the art which has been introduced in the introduction in Section 1.3.1. It is therefore divided into architecture modeling languages, domain specific languages, exploration approaches and development methodologies.

In Chapter 4 we are describing the methodology for the exploration of E/E Architectures. Moreover, we show how our approach seamlessly integrates into a customized V-Model XT development process and how it integrates into the existing embedded system development methodology SPES.

Chapter 5 describes the E/E Architecture Viewpoint where we propose meta-models for Software and Hardware Architecture (including the description of variants for *Hardware Resource*s) and the deployment of software onto Hardware Architectures in order to describe an E/E Architecture.

In Chapter 6 we introduce a domain specific language which is capable of formalizing an E/E Architecture exploration problem in terms of constraints and objectives. We will furthermore provide specific constraint and objective patterns which are common in the automotive domain.

Chapter 7 describes the transformation of the domain specific language into SMT, the validation of constraints to find contradictions and exploration which find optimal solutions of E/E Architectures.

In Chapter 8 we are evaluating this thesis. On the one hand we are evaluating the methodology by comparing it to the state of practice of E/E Architecture development and on the other hand we give two automotive use cases where we successfully applied approach for industrial sized E/E Architectures.

Chapter 9 then concludes this thesis by summarizing it and giving an outlook on future work.

2. Automotive architectures and standards

In this chapter, we provide an overview over automotive Electric/Electronic architectures (E/E Architectures). We especially to focus on one of the most important standard that influences the development of such architectures: ISO26262 (International Standard Organization Norm 26262). ISO 26262 is the functional safety standard for passenger vehicles and an adaption of IEC 61508.

In Section 2.1, we have a closer look at ISO 26262 and standard compliant engineering.

In Section 2.2, we elaborate on model-based systems engineering which is, according to ISO 26262, one way to develop automotive systems. In this Section, we introduce the important aspects of MbSE covering the concept of Viewpoint/View and Component/Interface which are used throughout this thesis.

In Section 2.3, we have a look at state-of-the-art of E/E Architectures and at future trends, as E/E Architectures are eventually evolving from federated, distributed to centralized architectures.

2.1. ISO 26262 compliant engineering

In order to develop safe passenger cars, the development has to comply to the international safety standard ISO 26262. This standard proposes a certain development process which an automotive company has to adhere to (Section 2.1.1). Furthermore, this standard introduces a component model to describe E/E Architectures (Section 2.1.2) easing the safety classification of E/E Architectures in terms of automotive safety integrity levels (ASIL) (Section 2.1.3).

2.1.1. Development process

The engineering of automotive E/E systems is driven by the development of safe products. Thus, development processes of automotive systems must adhere to ISO 26262.

"ISO 26262 is intended to be applied to safety-related systems that include one or more E/E systems and that are installed in series production passenger cars with a max gross weight up to 3,5 t" [25]. It "is the adaptation of IEC 61508 [(International Electrotechnical Commission Norm on Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems) [26]] to comply with needs specific to the application sector of E/E systems within road vehicles" [25]. Hence, it "applies to all activities during the safety lifecycle of safety-related systems comprised of electrical, electronic, and software elements that provide safety-related functions" [25].

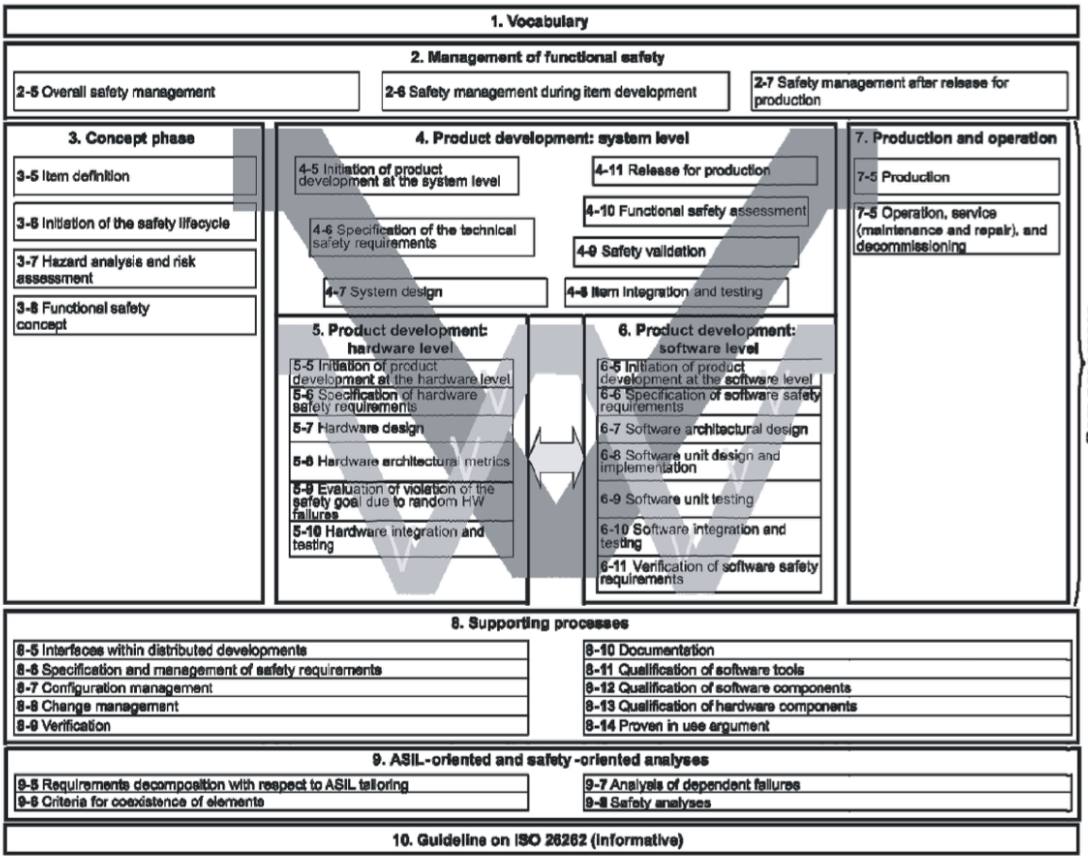


Figure 2.1.: Automotive system development according to ISO 26262 [25]

As illustrated in Figure 2.1, ISO 26262 uses the V-Model as a basis to describe the development process focusing on the safety aspects of passenger cars. The following Chapter statements refer to ISO 26262. In Chapter *4.7 System design* the development is divided into *5. Product development on hardware level* and *6. Product development on software level*. Chapter *4.8 Item Integration and Testing* is requiring the integration of both software and hardware level as well as testing. Figure 2.2 schematically depicts this process.

In this way, ISO 26262 enables to divide the system into sub-systems, which can be developed by suppliers. In the end, all sub-systems are composed again.

2.1.2. ISO 26262 component model

Figure 2.3 illustrates the component model proposed by ISO 26262. A system can be composed of n (sub-) systems each of which implements a set of functions m. An Item is

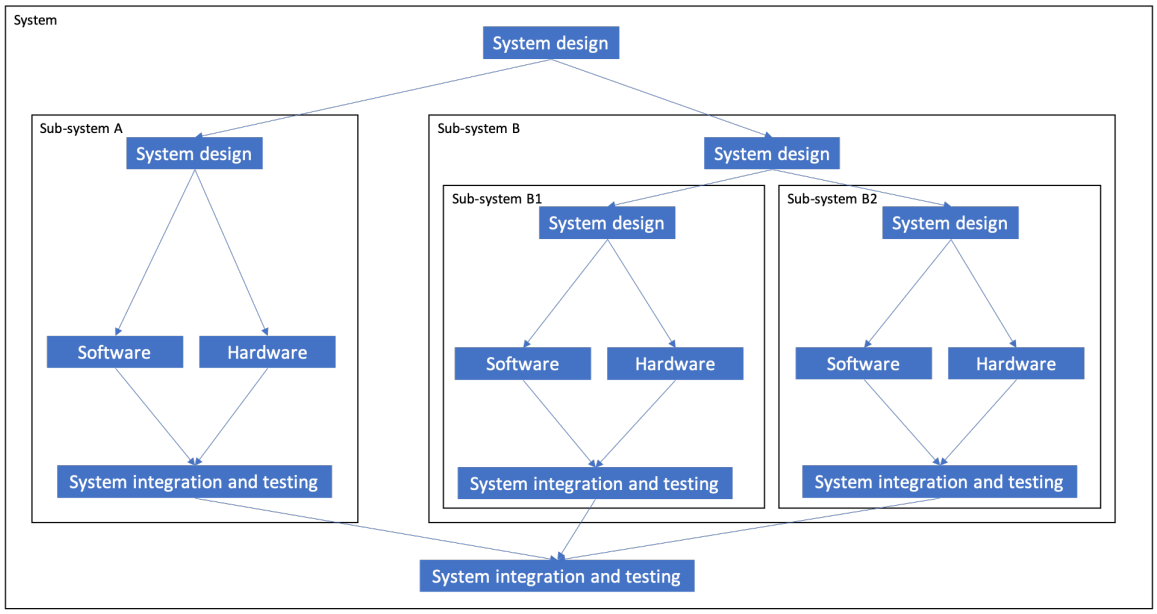


Figure 2.2.: System design example taken from ISO 26262 [25]

thereby defining a system or (sub-) systems of consideration. A system is composed of n components. A component, similar to the system, can be composed n components. Thus, it is possible to model a hierarchic component model. An atomic component which is not further hierarchically decomposed is either a software part of a hardware unit. Lastly, an element describes a sub-set of (sub-) systems, components, software parts and hardware units.

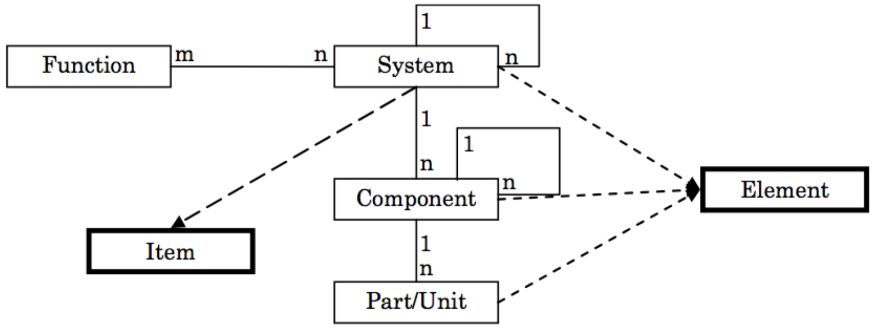


Figure 2.3.: ISO 26262 component model [25]

Figure 2.4 gives an exemplary overview how an automotive system is composed using the component model introduced above. Hence, a system consists of *E/E Components*, *Communication* and *Other technology Components*. An *E/E Component* is defined as either a *Sensor*, a *Controller* or an *Actuator*. Each component can be split into *Hardware Parts* and *Software Units* which are the atomic parts in the component model.

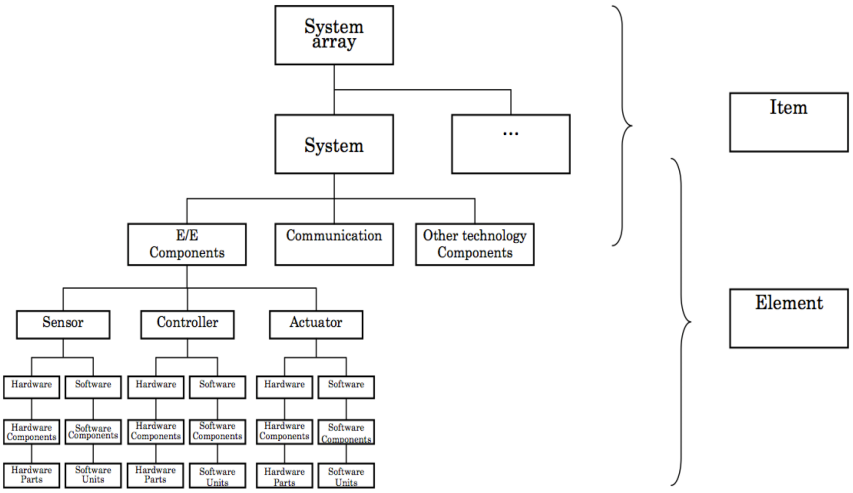


Figure 2.4.: ISO 26262 component model in system development [25]

2.1.3. Automotive Safety Integrity Level

According to ISO 26262, the functional safety has to be classified. In order to do that, there exist Automotive Safety Integrity Levels (ASIL) from A to D. Hereby, ASIL A is the lowest rating in terms of risk protential and ASIL D the highest. They act as risk reduction requirements for a certain functionality. Unlike IEC 61508, which proposes the Safety Integrity Level (SIL), they are not defined probabilistically. Table 2.1 shows how an ASIL is determined.

The ASIL rating is determined according to

- the severity S (1-3), denoting the potentially caused damage,
- the exposure E (1-4), denoting the likelihood a driver will encounter such a situation,
- the controllability C (1-3), denoting how easy it is for the driver to handle the situation.

For example, considering a windshield wiper malfunction on the highway during heavy rain may have life-threatening severity (S3) if an accident happens. Regarding the exposure,

		Controllability		
Severity	Exposure	C1 (simple)	C2 (normal)	C3 (difficult)
S1 (Light/moderate)	E1 (very low)	QM	QM	QM
	E2 (low)	QM	QM	QM
	E3 (medium)	QM	QM	A
	E4 (high)	QM	A	B
S2 (Severe)	E1 (very low)	QM	QM	QM
	E2 (low)	QM	QM	A
	E3 (medium)	QM	A	B
	E4 (high)	A	B	C
S3 (Life-threatening)	E1 (very low)	QM	QM	A
	E2 (low)	QM	A	B
	E3 (medium)	A	B	C
	E4 (high)	B	C	D

Table 2.1.: ASIL determination

heavy rain in a middle European country could be rated high (E4). The controllability of such an event would be difficult as the driver could not see anything and would thus not be able to control the car anymore (C3). As a consequence, the windshield wiper would have to be developed as an ASIL D item.

Considering the component model of Figure 2.4, the ASIL would propagate from the item to the element level. Hence, the software part(s) and hardware unit(s) have to be developed, taking into account the items ASIL. In the example, the windshields wipers software and hardware would have to be developed according to ASIL D.

2.2. Foundations in model-based systems engineering

Model-based Systems Engineering (MbSE) is one way to develop systems. It is an engineering approach that, opposed to document-based approaches, provides formalisms which enable frontloading techniques. As MbSE abstracts away from resource specific implementation issues, it enables techniques such as analysis, synthesis or simulation already at early stages of development. ISO 26262 also encourages the use of MbSE especially for engineering software units of automotive systems [25].

In the following, we will have a closer look at the general concepts of MbSE in Section 2.2.1. We furthermore introduce two of its key aspects, namely the concept of viewpoints and views in Section 2.2.2 and the concept of components and interfaces in Section 2.2.3.

2.2.1. Model-based systems engineering (MbSE)

The International Council of System Engineering (INCOSE) is a non-profit organization whose mission is "To address complex societal and technical challenges by enabling, promoting, and advancing systems engineering and systems approaches." [1]. They define Model-Based Engineering as

> "An approach to engineering that uses models as an integral part of the technical baseline that includes the requirements, analysis, design, implementation, and verification of a capability, system, and/or product throughout the acquisition life cycle." [27]

Their definition is based on the final report of the Model-Based Engineering Subcommittee of the (US) National Defense Industrial Association (NDIA) [28].

Hence, this approach can replace document-based development approaches using system models during all activities throughout the development life-cycle. Furthermore, as models provide an abstraction of the system, they abstract away from resource specific implementation issues and, because of their fiormal character, enable a variety of automations, at early stages of development. Among others, this has the following advantages:

- Simulation
  Simulation is the execution of models that are described using a behavioral specification. Simulation is used in early design phases supporting the understanding of (composed) models, namely the desired (sub-) systems behavior, independent from the their implementation and technical realization. Thus, this can be seen as an early stage systems test.

- Analysis
  Because of the formal nature of models, (sub-)systems can be checked for correctness. Model checking [29], for instance, is an automated verification technique, which can prove the correctness of models and thus the correctness of the system, w.r.t. certain properties (e.g. formalized requirements), itself. Hence, (sub-)systems can be formally verified already at early stages of development.

- Synthesis
  A model-based approach enables the synthesis of new models out of existing ones. Given a formal model of requirements and a model of the system, synthesis methods enable the automatic generation of further system artifacts (e.g. by using mathematical solvers). One example is the automatic generation of (automotive) deployments from software components to hardware components. (In this thesis, we use the word exploration as a synonym for model synthesis.)

In particular, this thesis covers the model synthesis aspect. This means the automatic generation of automotive E/E Architectures given a set of formalized requirements.

[1] `https://www.incose.org/about-incose` (accessed 2020-05-05)

2.2.2. Viewpoints and Views

The concept of Viewpoint and View can be found in the ISO-42010 standard [30] on architecture description in systems and software engineering. Figure 2.5 depicts an overview

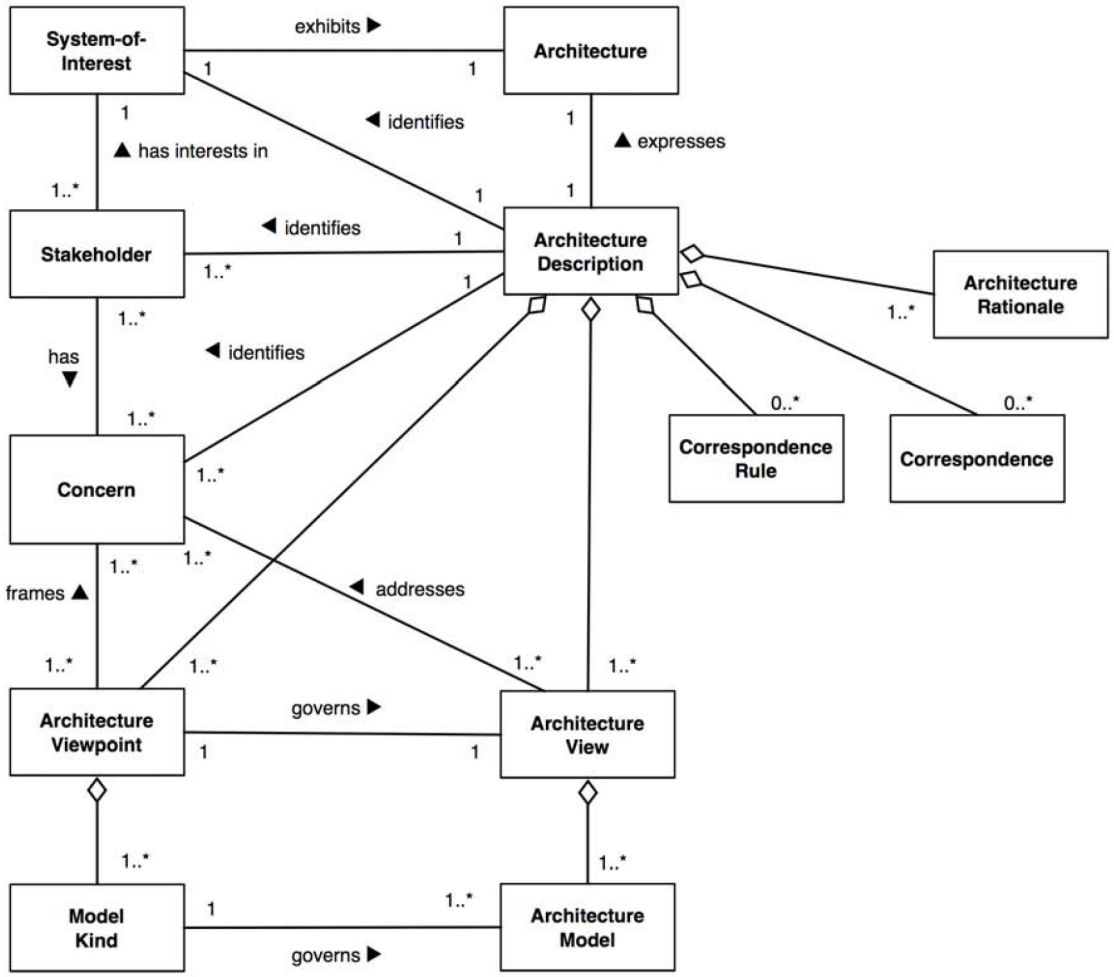


Figure 2.5.: Architecture description and its relation to viewpoints and views according to ISO 42010 [30]

over Viewpoints on Architecture Descriptions. An *Architecture Viewpoint* frames certain *Concern*s of specific *Stakeholder*s considering a *System-of-Interest*. An *Architecture Viewpoint* governs an *Architecture View* which consists of an arbitrary number of *Architecture Model*s and which address the *Concern*s of certain *Stakeholder*s.

Considering, for example, the E/E Architecture of an automotive system, there might be a dedicated architecture viewpoint which is the *Concern* of a system architect. Considering

an exploration of these E/E Architectures, there is a need for additional viewpoints focusing especially on formalization and automation aspects.

In short, a viewpoint can be described as the standpoint from which one is looking at a system of interest and the view in turn is what one sees. The concept of viewpoint can moreover be found in the ISO- standard which provides a general reference model for open distributed processing (RM-ODP) [31].

Model-based design methodologies propose to divide a system under development into different viewpoints in order to cope with the rising complexity of today's embedded system development. A viewpoint describes the system from the perspective of a specific stakeholder. A view in turn describes the models and languages used to describe a specific viewpoint. SPES 2020 [16] and the EAST-ADL2 [12] [32] are examples of such development methodologies which are using this approach.

One of the advantages of this approach is that the system's functionality can be developed independently from the hardware realization. In the SPES 2020 methodology, for example, this is realized by a logical and technical architecture of the system. In the EAST-ADL2 it is realized by a Functional Design Architecture and a Hardware Design Architecture.

2.2.3. Components and Interfaces

The notion of components which exchange information via interfaces is a wide spread theory of computation. It is opposing a classical object oriented approach as it is allowing the concurrent modeling of interaction in contrast to sequential interaction and is based on the exchange of information rather than exchange of control [33]. Broy et al., for example, describe an algebraic theory over this model of computation [34, 35]. Using such a theory to define the semantics of models enables well-defined operations on those components.

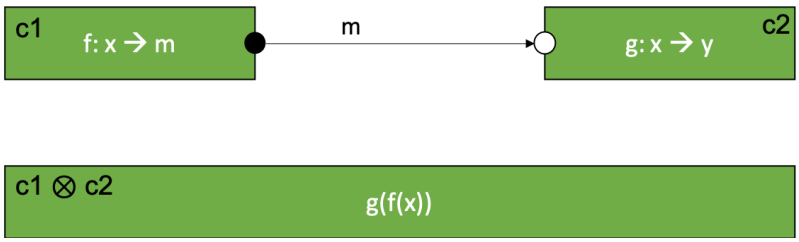


Figure 2.6.: [34]

One important aspect is the concept of compositionality. Given two components c1 and c2 (Figure 2.6 top) which execute two functions f and g and exchange message m which is the computational result of c1, the composition of c1 and c2 ($c1 \otimes c2$, Figure 2.6 bottom) is equivalent given the semantics proposed by Broy [34].

This fact is especially useful, due to the fact that automotive systems are usually developed by different suppliers each of which develop a certain part (sub-system) of the system. If the whole system (consisting out of an arbitrary number of sub-systems, similar to c1

and c2) is formally described using components and interfaces with the described semantics, while each supplier develops one sub-system, one can give mathematical evidence that the composition of all sub-systems is correct.

2.3. E/E Architectures

Building upon the component model proposed by ISO 26262 (cf. Section 2.1.2; Figure 2.4), we will define an automotive E/E Architecture as a topology network composed of *E/E components* (e.g. Sensors, Controllers or Actuators) which are connected by *Communication* components (e.g. buses). Considering the definition (of ISO 26262) of an *E/E component*, the architecture can be divided into software and hardware. In this way, hardware parts together with *Communication* components (e.g. bus systems like CAN) form the hardware topology of the system, whereby each hardware part can execute specific software units. The software units can exchange information realized by messages which are sent via the *Communcation* components. The E/E Architecture is thus controlling the mechanical parts of a vehicle.

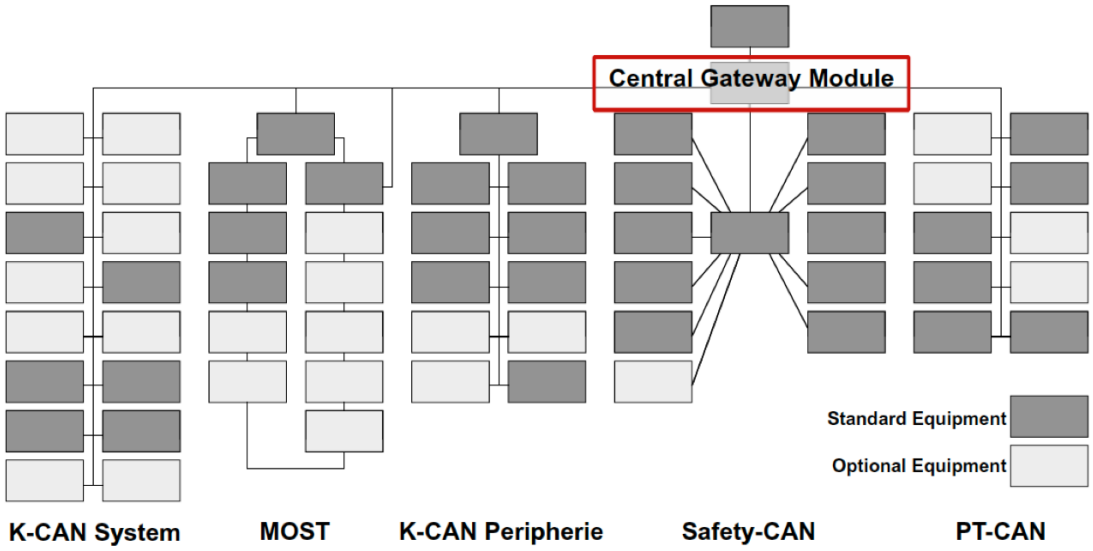


Figure 2.7.: E/E Architecture of a BMW 7-series in 2005 taken from Zeller et al. [2]

Figure 2.7 schematically shows such an exemplary E/E Architecture. The *E/E components* are visualized as rectangles and and the *Communication* components, in form of buses, is visualized as lines between the *E/E components*. Thus the software units which are part of an *E/E component* can exchange messages through these connections.

2.3.1. State-of-the-art

Since the introduction of the first Electronic Control Unit (ECU) in 1968 [36], the E/E Architecture has evolved as a federated, distributed system. Over the years, new functionality was added in a bottom up manner introducing new ECUs which had to be connected to each other to exchange information [37]. Considering the structure of the automotive industry, this not only encouraged but enabled to optimize a system of OEMs (Original Equipment Manufacturer) and Suppliers, where a supplier is responsible for the development of a distinct ECU and its software which is in the end integrated by the OEM. However, as those architectures were never designed or re-designed in a top-down manner [37], the E/E Architecture evolved into an ever growing distributed system, which is hard to maintain and difficult to test. One of the biggest difficulties is the non-determinism of communication in such systems as an efficient communication via the bus systems can only be achieved by a significant amount of multiplexing [1].

Figure 2.7 illustrates an exemplary E/E Architecture of a BMW 7-series in 2005, consisting of 62 ECUs, taking into account standard and optional equipment. Those ECUs are connected via different bus systems like the Safety-CAN for safety related functionality or MOST which is often used for multimedia applications as it provides a high throughput. One can easily see, that the E/E Architectures shows a typical matrix organization projecting the organization structure of an OEM as mentioned above. Taking into account that in 2010 such architecture was already consisting of up to 100 ECUs [2][3], they may not be optimal considering that each newly introduced ECU rises the complexity in terms of communication and possibly introduces new defects. Especially, with the emergence of driver assistance functions and eventually autonomous driving functions, today's E/E Architecture cannot cover the upcoming requirements of those features in an efficient manner [38].

2.3.2. Trends

As future E/E Architectures will have to carry an ever increasing amount of software implementing a variety of functions the trend of E/E Architectures goes from federated, distributed towards centralized architectures. In 2006, Broy already envisioned the future of the E/E Architecture as a more centralized architecture with only few dedicated ECUs [37]. Kugele et al. propose Service Oriented Architectures (SOA) as one way of reducing the complexity of developing E/E Architectures going away from the traditional ECU-based development [39], which would also encourage a centralized E/E Architecture. In the following, we want to have a closer look at the current trends E/E Architecture development from an industrial and research perspective.

According to Traub et al. (BMW), the development will no longer be focused on locally optimizing an ECU but rather system-level optimization driven by system architects [40]. Going away from the sender- receiver communication paradigm in today's E/E Architectures, future communication will be handled by a central communication server [40].

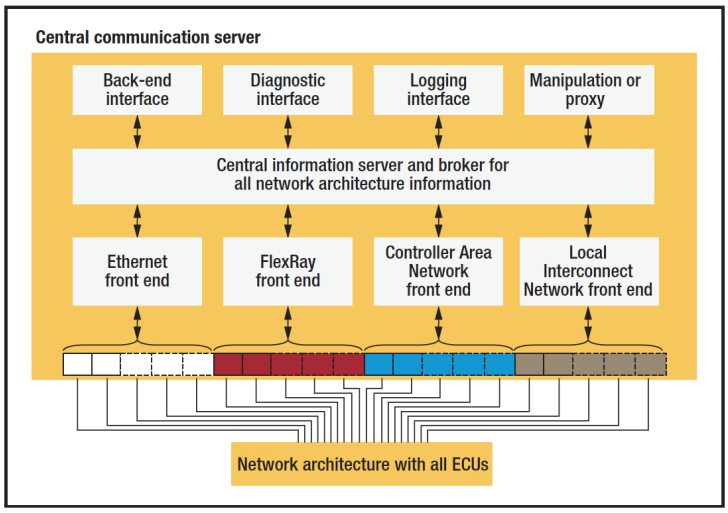


Figure 2.8.: E/E Architecture realized with a central communication server [40]

Figure 2.8 schematically shows such an architecture where a central information server is connected to the cars ECUs via different bus systems such as Ethernet, FlexRay, CAN and LIN. Handling not only the communication inside the vehicle through a central server, this architecture offers a variety of interfaces accessible through this server for, among others, diagnostics or logging.

Benckendorff et al. (Bosch) analyze the current and future E/E Architecture trends [5]. In terms of topology patterns, they are comparing ring, linear bus topology (backbone) and star topologies. They are arguing that ring topologies do not play a central role, today. Figure 2.9 shows the two main topology types: backbone and star. They are arguing that star architectures will play a central role in the future in order to handle the increasing amount of signals and their prioritization as well as security needs.

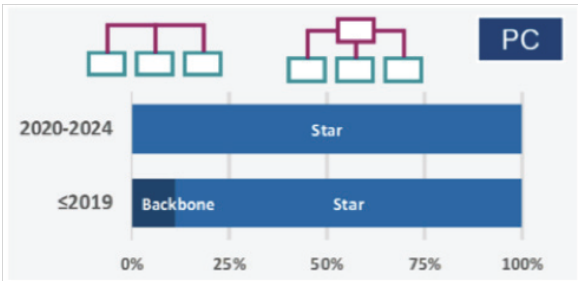


Figure 2.9.: Future E/E Architecture patterns [5]

Figure 2.9 illustrates the supposed star and backbone pattern evolution for passenger cars (PC) according to Benckendorff et al. In the next few years star topologies will thus

be the predominant pattern for E/E Architectures [5]. This corresponds to the trend of continuously centralized architectures.

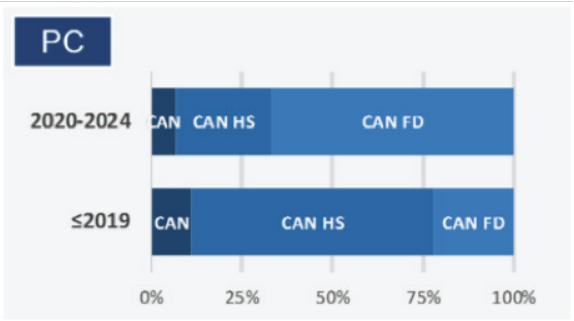


Figure 2.10.: Future use CAN bus types in E/E Architectures [5]

Due to the increasing network demand, they argue that the trend considering CAN buses goes towards CAN with flexible data-rate (CAN FD) which is shown in Figure 2.10. Additionally, Ethernet will also be increasingly used within E/E Architectures, because of its high throughput.

Sommer et al. [4] and Büchel et al. [41] present a vision of future in-vehicle architectures from a research perspective. They are referencing respective standards in avionic like IMA (Integrated Modular Avionics [6]) which propose the existence of a centralized vehicle control-computer. Their so called RACE architecture thus consists of a centralized computer which executes high level functionality. This computer is connected to few ECUs (which are called vehicle control-computers) responsible for controlling and collecting data from sensors and actuators of the vehicle. Figure 2.11 schematically shows the RACE architecture where the central vehicle control computer is connected to the actuator ECUs and sensors of the car.

Backed by frequent discussions with Continental, not only in the course of an ongoing collaboration in [21] and [23], there is a clear trend towards centralized E/E Architectures. So the big challenge is the transition from federated, distributed architectures to centralized architecture. Considering that the architectures represent organizational and OEM/supplier structures this transition can not be ad-hoc. Braun et al. also identified corporate structure as the most important barrier considering innovations of E/E Architectures [42]. Furthermore, cost, weight and quality are identified as the important criteria regarding future E/E Architectures. Hence, one of the most important industrial asked questions is how to gradually develop more and more centralized E/E Architectures while controlling cost, weight and quality.

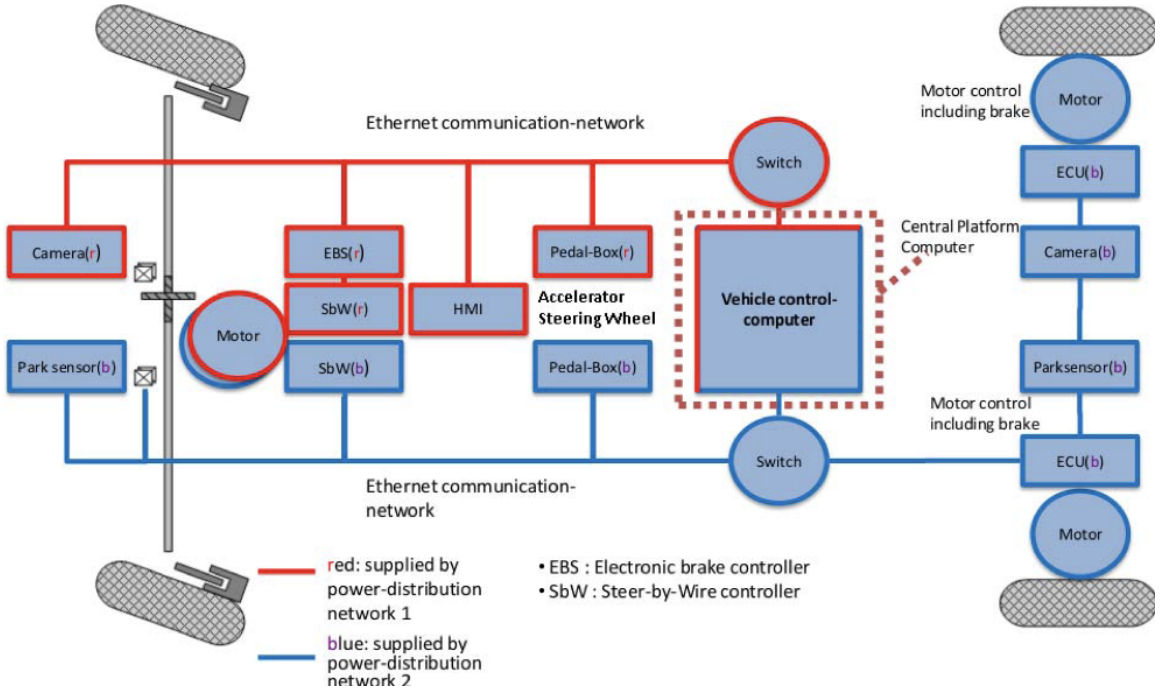


Figure 2.11.: E/E Architecture of the RACE car [4]

3. Related work

This chapter summarizes the related work of this dissertation. Section 3.1 describes existing architecture description languages. Section 3.2 describes domain specific languages which are relevant in the context of this thesis. In Section 3.3 existing work related to design space exploration aspects is described. As this thesis focuses especially on solving exploration problems, deployment, topology and generic synthesis approaches are described in this Section. Lastly, Section 3.4 covers two industrial and two related academic model-based development methodologies.

3.1. Architecture Description Languages (ADL)

There exist a variety of architecture modeling languages both from academia as well as industry. In the following, we will present some of theses architecture description languages (ADLs) which are related to this thesis, are originally from both domains and could thus be used to model an E/E Architecture.

3.1.1. UML/SysML

The SysML (Systems Modeling Language [9]) is modeling language derived from the UML 2 (Unified Modeling Language [8]). It reuses most of the concepts of UML 2 and provides additional extensions in order to provide a systems engineering language. The goal hereby is to "support[s] the specification, analysis, design, verification, and validation of a broad range of complex systems. These systems may include hardware, software, information, processes, personnel, and facilities.". [9]

One of the main concepts in the SysML is a *Block*. A *Block* inherits from a UML Class and is used to specify hierarchies and interconnections. Its purpose is to define a collection of features to describe a system or other elements of interest (Hardware, Software, Data, Procedure, etc.). [9]

SysML*Port*s are used to describe the interface of a *Block*. They can also by typed by a *Block*. The SysML also provides the more specific concepts of *Proxy Port*s and *Full Port*s. ("Proxy ports define the boundary by specifying which features of the owning block or internal parts are visible through external connectors, while full*Port*s define the boundary with their own features." [9])

SysML *Part*s are used to describe the internal structure of a *Block*. They are typed by *Block*s and can therefore be regarded as an instance of a specific *Block*. The interconnection between *Part*s or*Port*s can be realized by *Connector*s (e.g. BindingConnector,

BidirectionalConnector, UnidirectionalConnector). The internal structure of a *Block* can thus be described through different *Part*s (which are typed by different *Block*s) exchanging information through*Port*s attached to those *Part*s and connected via *Connector*. Figure 3.1 exemplary depicts the internal structure of a *Block* by using a internal block diagram (ibd).

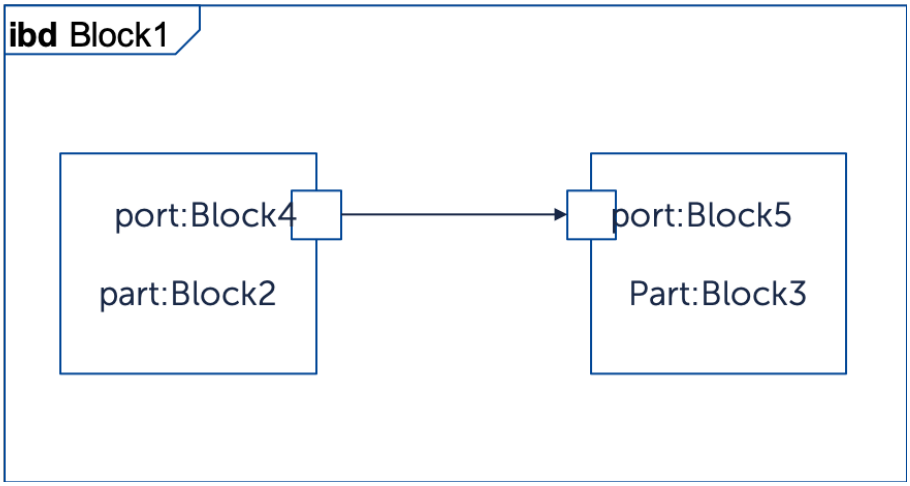


Figure 3.1.: Internal block diagram of a SysML *Block* 'Block1' which contains two *Part*s which are connected with a *Connector* between two *Port*s attached to the *Part*s. The colon separates the name (left-side) from the type (right side) of the respective model element. Both *Part*s and *Port*s can thus be typed by *Block*s.

The UML is a general purpose modeling language which mainly targets software modeling and does not include any hardware specific aspects needed for modeling E/E Architectures. The SysML has been built to enable systems modeling. However, it also does not provide any E/E specific hardware models which enable modeling E/E Architectures and it does not support variant modeling and thus does not support modeling variability aspects.

3.1.2. UML Marte

Marte (Modeling and Analysis of Real-Time and Embedded Systems) is a UML profile extending the UML with concepts for developing real-time embedded systems with the focus on performance and schedulability analysis [43][10]. Figure 3.2 schematically depicts the structure of Marte. Refining a Generic Resource Model, Marte consists of a design model on the one hand, and an analysis model on the other hand.

The Marte design model distinguishes between Software Resource Modeling (SRM) and Hardware Resource Modeling (HRM). The SRM provides modeling elements such as semaphores and task in order to model the software of a real time system. By using special stereo-

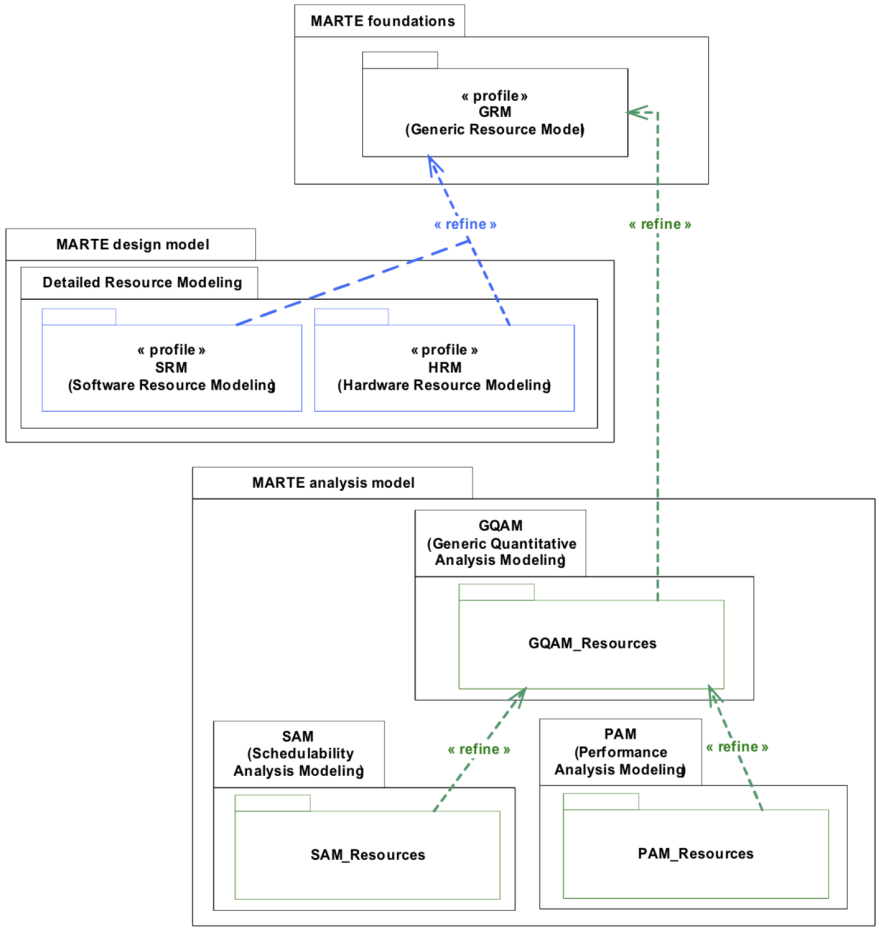


Figure 3.2.: General structure of UML Marte taken from [43]

types SwSchedulableResource or MessageComResource a UML Class can be described as schedulable resource like a task or as a message which is exchanged between communicating resources.

The HRM provides modeling elements to describe the hardware execution platform. It furthermore restricts the allocation relationship such that only application elements can be allocated to platform model elements [43]. By using special stereotypes like e.g. *HwProcessor* or *HwBus* a UML Class can be described as a computation resource (comparable to a ECU) or a communication resource (comparable to a bus).

UML Marte distinguishes between different hardware elements (processor, bus) and provides deployment models. However, it also does not support variant modeling and thus does not support modeling variability aspects.

3.1.3. AUTOSAR

AUTomotive Open System ARchitecture (AUTOSAR) is a development partnership among Automotive OEMs and suppliers. The core partners - which also founded the partnership - BMW Group, Bosch, Continental, Daimler, Ford, GM, PSA Groupe, Toyota and Volkswagen state, that "[t]he objective of AUTOSAR is to establish an open industry standard for the automotive software architecture between suppliers and manufacturers"[11].

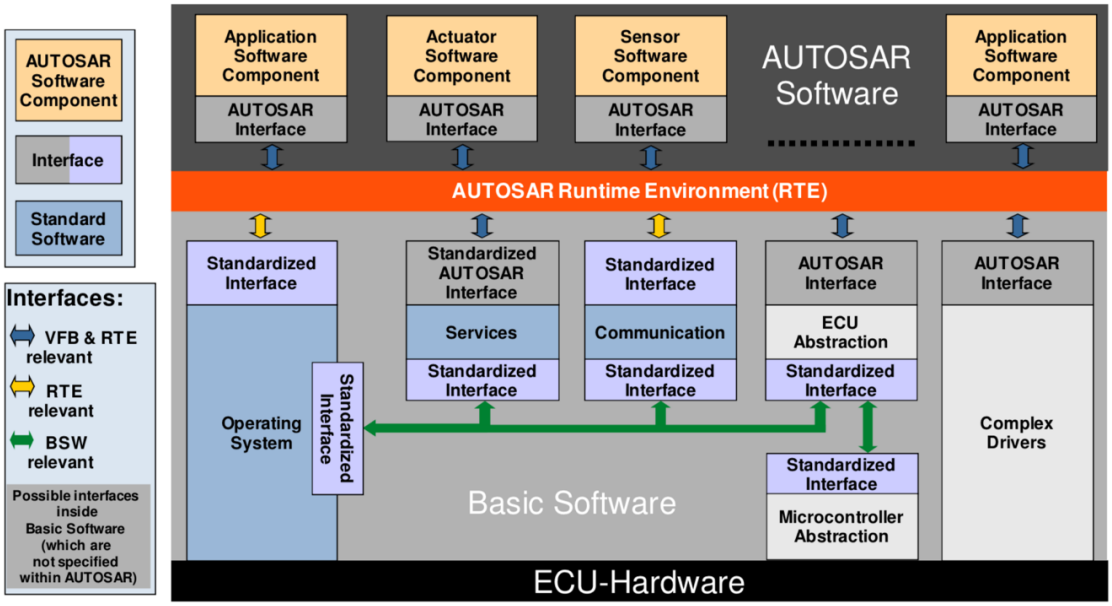


Figure 3.3.: AUTOSAR layered architecture taken from [44]

Figure 3.3 illustrates the layered architecture of AUTOSAR. Basically, it consists of three Layers which run on an electronic control unit (ECU).

The AUTOSAR Software Layer consists of Software Components that are modeled in a typical component based fashion.

The AUTOSAR Runtime Environment (RTE) provides communication services to the Software Layer abstracting away from the hardware specific implementations and thus making the Software Components independent from the ECU. The RTE enables communication within or between ECUs.

The Basic Software Layer which covers the lower hardware level implementations. Based on the specific ECU-Hardware it provides an interface for the RTE to e.g. the operating system, lower level communication implementation or drivers.

Although this modeling approach especially targets E/E Architectures, it focuses on implementation specific aspects which makes it unsuitable to use in early stages of development (frontloading). Furthermore, it does not provide variant modeling capability.

3.1.4. EAST ADL2

EAST-ADL2 (Electronics Architecture and Software Technology - Architecture Description Language 2) is an architecture description language which was developed in the course of the ITEA EAST-EEA and ATESST project [12]. The purpose of this to build a a high level abstraction up on AUTOSAR. As AUTOSAR is providing detailed models on implementation levels, the goal of EAST-ADL2 is to cover, among others, higher level requirements, behavior, software and hardware models. Therefore, it provides a basis for documentation and management of system models on various abstraction levels [12].The EAST-ADL2 can be used through a UML2 profile which specifies this domain specific language.

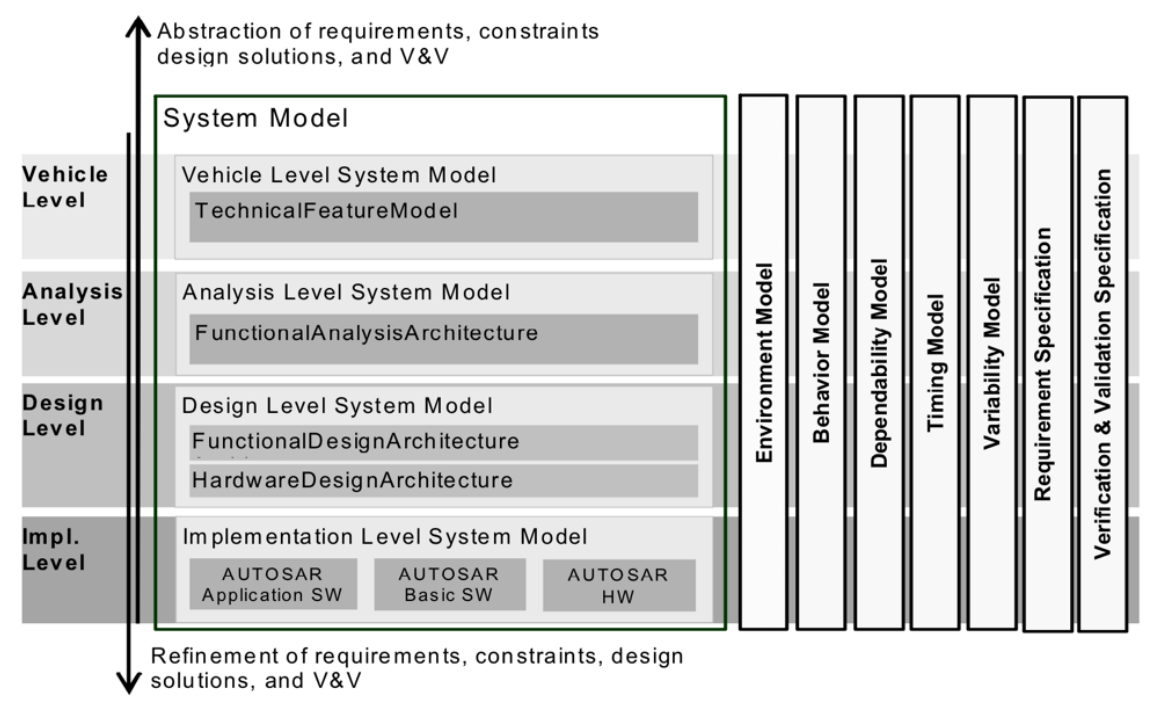


Figure 3.4.: EAST ADL2 structure taken from [45]

Figure 3.4 shows a schematic overview over EAST-ADL2. Where the Implementation Level is modeled and described using AUTOSAR consisting out of Application Software the Basic Software and Hardware, the EAST-ADL2 extensions are described on higher abstraction levels (Design, Analysis and Vehicle Level). On each of these levels different additional models can be described. They are described orthogonal to the abstraction levels in Figure 3.4 (Environmental Model, Behavior Model, Dependability Model, ...). Variability modeling is one of them, allowing to introduce variants on every level of those abstractions.

On the Design Level, the *FunctionalDesignArchitecture* and the *HardwareDesignArchitec-*

ture are defined. This means, that the hardware topology and its resources can be defined as well as the functions which will be allocated onto the hardware resources. An E/E Architecture exploration could thus be performed on the design level of EAST-ADL2.

However, the design level is missing a proper consideration of the software architecture as it is only regarding a *FunctionalDesignArchitecture*. This means that functions are directly deployed to the hardware design architecture. So the definition of an E/E Architecture on the design level is missing the software aspect.

3.1.5. AADL

"In November 2004, the Society of Automotive Engineers (SAE) released the aerospace standard AS5506, named the Architecture Analysis & Design Language (AADL). The AADL is a modeling language that supports early and repeated analyses of a system's architecture with respect to performance-critical properties through an extendable notation, a tool framework, and precisely defined semantics [13].

The language employs formal modeling concepts for the description and analysis of application system architectures in terms of distinct components and their interactions. It includes abstractions of software, computational hardware, and system components for (a) specifying and analyzing real-time embedded and high dependability systems, complex systems of systems, and specialized performance capability systems and (b) mapping of software onto computational hardware elements." [13]

The AADL distinguishes between the application software on the on hand, and execution platform on the other hand. A system in the AADL is thus a composite out of application software and execution platform.

The application software is composed out of the following elements.

- **Thread**: Active component that can execute concurrently and be organized into thread groups.
- **Thread group**: Component abstraction for logically organizing thread, data, and thread group components within a process.
- **Process**: Protected address space whose boundaries are enforced at runtime.
- **Data**: Data types and static data in source text.
- **Subprogram**: Concepts such as call-return and calls-on methods (modeled using a subprogram component that represents a callable piece of source code).

The execution platform is composed out of the following elements:

- **Processor**: Schedules and executes threads.
- **Memory**: Stores code and data.

- **Device**: Represents sensors, actuators, or other components that interface with the external environment.
- **Bus**: Interconnects processors, memory, and devices.

Although the AADL supports modeling hardware architectures and software architectures and also deployment aspects, it has been built for the aerospace domain and is thus not properly suited for modeling automotive E/E Architectures. Moreover, it does not support variant modeling.

3.2. Domain specific languages

Considering the formalization of constraints, there exist only a few languages in literature. In particular, there is only one domain specific language which is used to describe constraints over (UML) models which is the Object Constraint Language introduced in Section 3.2.1. A language like SMTlib, introduced in Section 3.2.2, is not explicitly targeted at defining constraints over models and bound to the usage of SMT Solvers.

3.2.1. Object Constraint Language (OCL)

The object constraint language (OCL) [14], which is part of UML, allows not only to express constraints over UML models but also queries, manipulation and specification requirements, model transformations, well-formedness rules and code generation templates. By that it has become a default language for any type of constraint-based model driven engineering [46]. OCL is defined as a general purpose formal language complementing the UML. Its most important types of expressions according to Cabot et al [46] are:

- **Invariants**: specify conditions a system has to comply with.
- **Initialization expressions**: specify initial values of objects upon creation.
- **Derived Elements**: similar to initialization expression, but constraining the values of an objective throughout its life span.
- **Query operations**: similar to database queries, which query a model and return information.
- **Operation contracts**: specify pre- and postconditions of the system after the execution of an operation.

Although this language allows to define constraints in the form of invariants, it does not support the formalization of optimization objectives. Furthermore, it is a very comprehensive and complex language which is general purpose by design and thus only specific to the domain of models in general.

Based on OCL, Maoz et.al [86], e.g., introduce a framework to verify extra-functional properties in component and connector models. Although, they provide language constructs which are similar to few of the constraint patterns which will be introduced in Chapter 6, it is rather targeted at verification than exploration of models.

3.2.2. SMTLib

SMT-lib2 is a language standard supported by most SMT solvers. "Satisfiability Modulo Theories (SMT) is an area of automated deduction that studies methods for checking the satisfiability of first-order formulas with respect to some logical theory T of interest" [47, 15, 48]. In particular a theory T does not have to be finite which distinguishes SMT from general automated deduction [48].

SMTlib is a international initiative to standardize the language of SMT solvers. Among others, two of the most well known SMT-solvers are Z3[1], which is developed by Microsoft, and yices[2] (current version yices2) which is developed at Standford Research Institute. In this thesis, we will focus especially on z3, as this SMT-solver additionally provides an optimization extension [49].

SMTLib provides a a low-level formalism which is bound the use of SMT solvers, is not targeted at models and does not cover automotive domain specific aspects.

3.3. Model exploration/synthesis approaches

There exists a variety of model synthesis approaches in literature. (In this thesis we use the word exploration to talk about the generation of models out of existing models. In this Section we synonymously use the word synthesis as some of the literature uses this word.) Most prominently, the so called deployment or mapping problem has been well studied allocating certain elements on resources according to constraints. Usually those approaches propose to formalize a synthesis problem such that it can automatically be solved by solvers. In the following, we will elaborate on the main contributions of those approaches and how our work differs from them.

3.3.1. Deployment

There has been a variety of works conducted in the area of deployment synthesis often combined with schedule synthesis.

Schätz and Voss et al. [50, 51, 52] propose a joint synthesis of deployment and schedule for mixed critical, shared-memory applications in [50]. They enable the calculation of optimized deployments w.r.t memory of cores and criticality of tasks on the one hand, and

[1] `https://github.com/Z3Prover`
[2] `https://yices.csl.sri.com/`

on the other hand, optimized task and message schedules (considering a global discrete time base). The approach is formalized in SMT and executed by SMT-solvers.

In [51], Schätz et al. build up on this approach, especially in the context of the automotive safety norm ISO 26262. Furthermore, they address the challenges of integrating the approach into a development process using AUTOSAR and answering the question of how a middleware like SysGO's PikeOS may be used to implement such an approach. The approach is formalized in SMT and executed by SMT-solvers. Both approaches are based on the constraint-based model-transformation presented in [52].

Both approaches [50] and [51] optimize a deployment of software tasks to computing resources by multiple optimization criteria, considering different constraints and additionally calculating schedules for these deployments. However they are only considering a fixed hardware topology with no variability considerations. A similar approach is presented by Kugele et al. [53].

Becker and Voss [54] present the deployment problem in the context of fail operational systems is presented by Becker and Voss in [54]. They propose an approach where a graceful degradation is ensured in case of hardware system failure by pre-calculating deployments for possible failure scenarios of the system. Thus, they consider different degraded hardware topologies in order to reconfigure the deployment of SW-components and ensure a fault tolerant behavior of the system. The approach is formalized and solved using SMT.

This approach focuses more on the fail operational aspect of embedded systems by pre-calculating possible deployments to enable graceful degradation in case of a certain system failure. Thus, a fixed hardware topology and no variability and timing is regarded in this approach.

Ross et al. [55] describe the synthesis and exploration of multi-level, multi-perspective architectures of automotive embedded systems. They are considering a system model consisting of a feature model, a functional analysis architecture and a hardware architecture containing a device node classification communication and power topology. So, multiple layers (multi-layer) are considered. At the same time they are describing different perspectives (multi-perspective) for these layers such as variability, latency mass, and cost. Using this multi-level, multi-perspective approach they are synthesizing candidate architectures. They are showing the applicability of their work by using two uses cases: an automotive power window system and a central door locking system.

This work considers multiple layers in the system specifically the deployment of (software) functions or components to the hardware layer considering execution units, communication units and the power topology. However, even through, they are considering variability though all layers, they are considering a fixed hardware topology, varying only in the type or optionality of components.

Aleti et al. [56] propose "an adaptive approach for controlling parameter values of genetic algorithms" [56] in order to solve the deployment problem. Based on a component model they are directly solving the deployment problem without using a formal language in between.

Besides using genetic algorithms, their work is only calculating the deployment of software components onto an existing hardware architecture. So the hardware topology is not open to exploration and furthermore does not contain variable elements.

Wozniak et. al [57] propose two approaches, using either mixed integer linear programming (MILP) or genetic algorithms to solve not only the deployment problem but to also the partitioning, scheduling and ordering problem. Using AUTOSAR models as input they are directly formalizing the problem to be solved.

Although this approach takes into account scheduling, partitioning and ordering, the underlying hardware topology is fixed and cannot be changed during the exploration. Furthermore, hardware variability is not covered by the approach.

Campeanu et al. [58, 59] present an deployment optimization approach based on mixed integer nonlinear programming. They especially focus on the deployment on CPUs and GPUs within a single hardware platform. They are also directly formulating the problem without the use of an intermediate language.

As the approach only calculated deployment for a single hardware platform, the topology is not open to exploration, meaning that there are no hardware variants covered by this approach.

Leserf et al. [60] perform an architecture optimization based on SysML models. They are also directly formalizing, without an intermediate language, the constraint satisfaction problem (CSP) in order to automatically solve it.

Although they are taking into account variability of the hardware elements the hardware topology is not open to exploration as only some hardware elements can be exchanged with others or omitted.

Lukasiewycz et al. [61] a design space exploration of embedded platforms by also considering product line optimization. Their two staged approach aims at firstly selecting candidate platforms which are in a second step optimized according to e.g. cost. They are proposing their own optimization algorithm which can be included into a heuristic approach like evolution¡ry algorithms.

Although they are considering variability through a product line of embedded platforms they only optimize a single platform and the respective deployment. Moreover, the sequential two step approach may not lead two optimal results as optimization in step two cannot alter decisions which were taken in step one.

Metzner and Herde [62] present a approach to solve the task deployment problem of distributed architectures. The approach is optimizing the the deployment also taking into account the task scheduling. They are using a SAT-based approach together with a an included nonlinear integer optimization to solve the task scheduling problem.

Although this approach considers takes task models as an input when exploring the design space of possible deployments, the underlying hardware architecture is fixed and does not contain any variable elements.

Basten et al. [63] present the Octopus Toolset for the design space exploration of embedded systems. It ingrates Uppaal and CPN Tools among others and uses the Y-chart DSE method [64]. They are optimizing application (software) , platform (hardware) and the mapping (deployment) between application and platform independently form each other.

Although the approach integrates tools to optimize and simulate resulting exploration solutions, including schedule optimization the platform is considered interdependently from the deployment and the application such that the topology cannot be altered and explored.

Peng et al. [65] propose a deployment optimization approach based on evolutionary algorithms. They are using AUTOSAR models as input to calculate optimized deployments. They are also considering scheduling in their approach.

However, the underlying hardware architecture, onto which software components are deployed, is fixed and is thus not open to exploration. Furthermore, variability in the hardware architecture is not considered in this approach.

Graf et al. [66] propose a multi variant design space exploration approach. They are using an over specified (150%) application model (software) together with an architecture template (hardware) to calculate different variants of cars and set up a a so called "Baukasten" from which all variants are built up. They are formulating the exploration problem directly (without the use of any intermediate language) as an integer linear program (ILP) to be solved in a hybrid optimization technique using a Pseudo Boolean solver and evolutionary algorithms as proposed by Lukasiewycz et al. [67].

Although this approach is using variants in the application model (software) the architecture template (hardware) is fixed considering the topology and cannot be changed. The approach is furthermore directly formulated as an ILP problem with no intermediate language, which would make it independent from the applied solving technique.

Thiruvady at al. [68] are using an Ant Colony Optimization Problem together with a constraint programming approach to calculate the automotive component deployment problem. They are taking into account three types of constraints (memory, collocation and communication) which are directly formalized without the use an intermediate language.

Although this approach is solving the deployment problem, the hardware topology is fixed and cannot be explored. Thus, variants of different hardware elements are also not

considered.

Kang et al. [69] propose a framework capable of optimizing reliability-aware mappings (deployments) onto mixed critical, multi-core systems. Their approach also considers the timing of tasks when calculating mappings.

Although this approach is calculating optimized deployments which are considered safety and reliability a fixed topology of hardware resources is considered and no variants in the hardware resources.

Pohlmann et al. [70] are proposing different viewpoints on hardware platform modeling in order to calculate a safe deployment. They are furthmore introducing a hardware platform description language to model hardware platforms.

This approach provides different views on hardware modeling and how a safe deployment can be performed. There is no automation approach of the deployment provided, meaning that the hardware topology cannot be explored and neither is the variability of hardware resources considered.

3.3.2. Hardware Topology

Bajaj et al. [71] propose an optimized calculation of of cyber-physical system architectures in with focus on safety and reliability. Taking into consideration the reliability of the system by using its interconnection structure, they are synthesizing topologies considering an upper bound of system failure probability which has to be met while optimizing costs such as number and weight of components. They are using a integer linear programming (ILP) based algorithms to generate architecture solutions. They are evaluating their approach by applying it to a passenger aircraft use case selecting optimal architectures for power generation and distribution.

This approach calculates an optimal hardware network topology using different optimization objectives while at the same time regarding constraints which have to be met. Thus, this approach starts with a completely unfixed hardware topology calculating an optimal topology of hardware resources. However, they do not consider the deployment of software tasks which could (heavily) influence the topology considering the computation and communication resource usage. The authors are furthermore not considering variability.

Glass et al. [72] present a multi objective routing and topology optimization approach for networked embedded system is presented. They are deploying a network of communication processes to a set of resources which are themselves connected. The former can be regarded as a network of software tasks whereas the latter represents the hardware network topology. Their approach, which uses multi objective evolutionary algorithm (MOEA) approach to calculate solutions, especially focuses on optimizing the routing of the hardware network topology considering weighted routing hops. By that, they are optimizing the topology of

the whole network by considering the deployment of processes to resources and optimally routing the consequential communication need between resources.

Although this work considers the optimization of a network topology of an embedded system, the network itself is fixed. Thus, the topology of the system may only change in the in the usage or non-usage of resources. Variability of resources is not considered in this work.

Pasricha et al. [73], Pinto et al. [74], Zverlov et al. [75], Mantovani et al. [76] present different approaches in the area of System-on-chip topology exploration. The main difference here is that these approaches do not consider a distributed embedded system but rather one specific hardware element of this systems which is optimized. Thus, those approaches consider more detailed information of the hardware an the wiring. Such explorations are generating topologies in terms of cores and their connections to internal bus systems and while at the same time calculating schedules for these internal architectures.

3.3.3. Generic synthesis approaches

Peter and Givargis [77] propose an approach similar to our work, consisting out of a component model which is formalized using their CoDeL language which is then transformed into SMT to synthesize systems. Their language CoDeL is a component-based description language which on the one hand is able top express the system model in terms of components and interfaces and on the other hand the requirements of the system in terms of constraints which have to hold in order synthesis a valid system.

Although they evaluate their work by using a deployment problem, they provides a rather generic synthesis approach. In particular, domain specific aspects like the variability of system components are not included. Moreover the resulting architecture is not optimized during the synthesis but rather a valid system is calculated.

Jackson et al. [78, 79], Kang et al, [80] present approaches implemented in the FORMULA framework. The FORMULA framework [78, 79] offers a formal specification language and an automatic model synthesis. The framework provides a generic solution to encode and solve design space exploration problems using SMT. There is further theoretical work conducted considering the performance by introducing symmetry breaking predicates into the resulting SMT problem formulation [80].

Although the FORMULA framework is a powerful tool which enables the almost any type of design space exploration it is therefore very generic. This means, that a E/E Architecture exploration problem would have to be formalized manually. This requires a lot of in depth mathematical knowledge. Furthermore, despite offering a language and exploration possibility, there exists no possibility to define a system model (software architecture, hardware architecture). A system model can only be used as input for an exploration if it is manually formalized first using their language.

3.4. Engineering methodologies

In the following, we give an overview over existing engineering methodologies. Sections 3.4.1 and 3.4.2 will cover the two academic methodology SPES and CESAR whereas Sections 3.4.3 and 3.4.4 will introduce the two industrial methodologies IBM Harmony and the Autosar methodology. Those methodologies are not an exhaustive overview but cover the most relevant methodologies for this thesis.

3.4.1. SPES

The SPES (Software Platform Embedded Systems) methodology which was developed in the course of two consecutive German research projects [16, 81] funded by the German Federal Ministry of Education and Research. The projects included academic as well as industrial partners with the goal of developing an applicable methodology for the development of future embedded systems.

Figure 3.5 shows that the SPES methodology divides the system development into four viewpoints: requirements, functional, logical and technical viewpoint. Those viewpoints can be engineered on different levels of granularity.

Thereby, the methodology is based on the notion of components, their respective interfaces and the composition of components (compositionality), as described in Chapter 2.2.3. This concept enables the description of sub-systems on each granularity level, which are in the end again composed to form the specified system. Usually the logical architecture is used as a starting point for a new granularity level. The architecture has been built according to the functions identified in the functional viewpoint which were derived from the requirements in the requirements viewpoint. This entails that each logical component and its interfaces are implementing certain functions and as such also the requirements. By that, a logical component can be engineered on a new level of granularity, where it can be developed as a sub-system according to its functions and interfaces. This enables to split the development of the system over different teams inside a company, including suppliers. The key aspect is that as long as the a logical component is developed exactly according to its interface provided by functions and requirements the system can be easily composed in the end. Thus, this enables a modular and re-usable system development which are core aspects of the SPES methodology.

In the following, we will give a short overview over each of the four viewpoints.

Requirements Viewpoint The requirements viewpoint provides means to systematically capture the requirements of a system using models. This viewpoint distinguishes between 4 different types of requirement models: (1) Context models document the environment (=context) of the system but are not part of the system, (2) Goal models document the objectives of the different stakeholders, (3) scenario models document the interaction between the system under development and its context and (4) solution-oriented requirements documenting a solution specific description of behavior, operations and information structure of

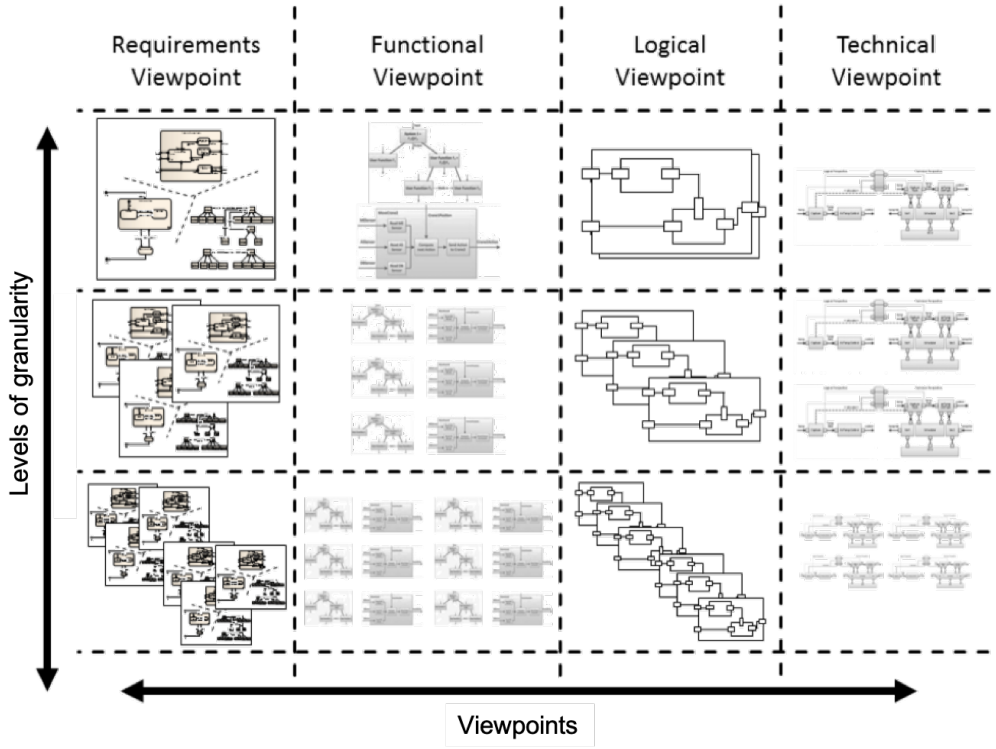


Figure 3.5.: The SPES matrix depiction the four viewpoints Requirements, Functional, Logical and Technical on the x-axis and the granularity levels on the y-axis. [16]

the system. Thus, the requirements viewpoint aims to gain a comprehensive understanding of the system easing the subsequent architecture steps.

Functional Viewpoint The functional viewpoint describes the behavior of the system derived from the (behavioral) requirements of the system. It provides two types of models: the functional black box and white box model. The functional black-box model translates the requirement models into a hierarchy of user function and their dependencies. In the functional white-box model, those user functions are further refined and already present an abstract solution of the respective functionality. Thus, the functional viewpoint aims to formally describe the system behavior and to improve the understanding of relations between different functions.

Logical Viewpoint The logical viewpoint describes the logical structure of the system in terms of a network of interacting components. It follows a typical component-based approach like e.g. proposed in [34]. A component in this logical architecture represents a certain functionality by processing the given inputs attached to the component and pro-

ducing a certain output. The design of the logical architecture depicting the logical view onto a system is driven by achieving maximum reuse and satisfying extra-functional requirements. On the one hand, this distinguishes this viewpoint from the the functional viewpoint (especially the functional white box model) where the focus lies purely on the functionality of the system. On the other hand, the logical viewpoint does not describe a technical realization of the system which is the difference to the technical viewpoint.

Technical viewpoint In the technical viewpoint, the system is described in terms of its platform/hardware specific aspects. The design of a technical architecture depicting the technical view onto a system, thus, covering the topology of hardware communication (e.g. bus systems like CAN) and processing resources (e.g. ECUs), on the one hand. On the other hand, this architecture details the specific aspects of these resources like e.g. the characteristics of multi-core platforms as opposed to single-core platforms, hierarchy aspects within the resources themselves or middle-ware aspects of the resources.

However, SPES is not providing a dedicated methodology enabling exploration in order to automatically calculate E/E Architectures.

3.4.2. CESAR

CESAR (Cost-efficient Methods and Processes for Safety-relevant Embedded Systems) was a European research project in the course of the ARTEMIS [3] joint undertaking which is a European public and private partnership in the field of embedded systems [17]. The goal of this project was to provide a component-based system development process. By that, they claim to have created a de facto European standard for embedded systems design [17]. They created the so called Reference Technology Platform (RTP) which serves as a generic model-based tool integration platform composed of (interoperable) tools, methods and processes.

CESAR is based on a model-based approach, using models throughout the development of a (embedded) system. Figure 3.6 schematically illustrates the structure of the proposed development during system design in the early stages of development referring to the left part of the V-Model.

The Operational Analysis aims to create operational scenarios of the system. Through the description of activity or use-case diagrams the interaction of the system is described with the environment. The Functional/Non-Functional Analysis refines those scenarios into formalized functional and non-functional requirements. Based on this analysis the logical and physical architecture of the system is defined. Multiple viewpoints are used here in order to capture the specific aspects of different stakeholders, working on the development of those architectures. The logical architecture is created by refining the functions of the preceding step into sub-functions and deploying them to logical components. The physical architecture is created representing the physical parts of the system. The logical components are then

[3] https://artemis-ia.eu/

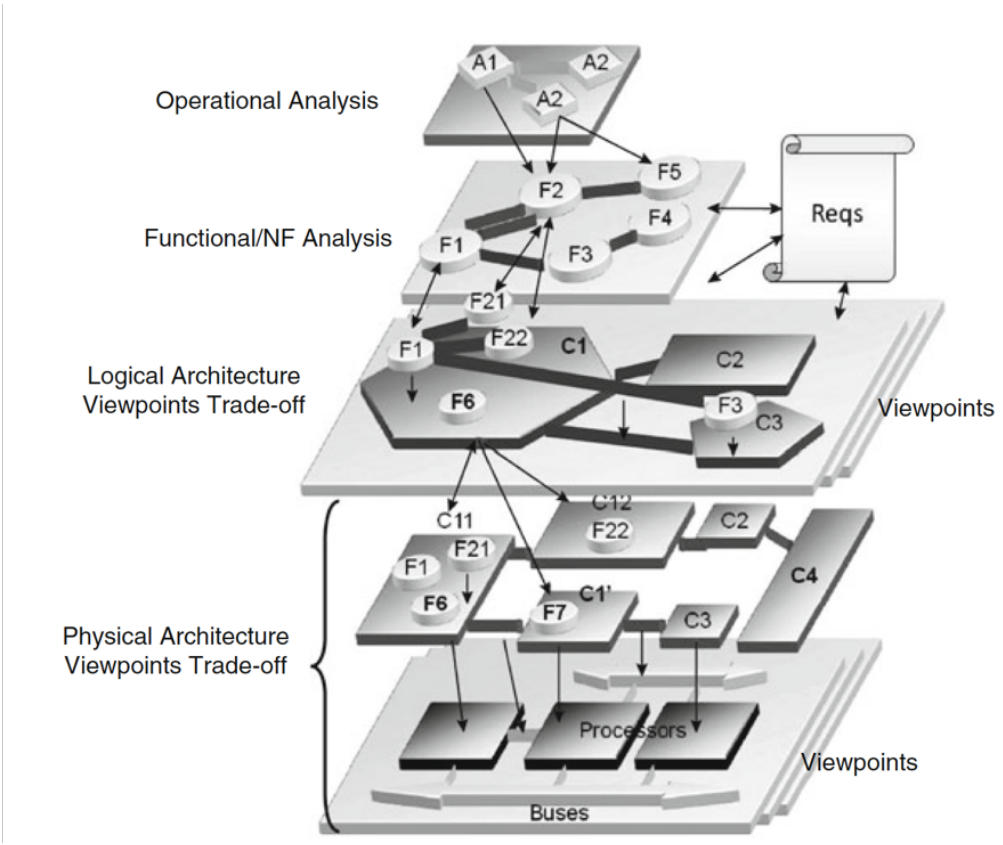


Figure 3.6.: System design in CESAR [17]

allocated to the physical components. CESAR explicitly mentions exploration methods which can be used to explore both logical and physical architectures and to explore possible deployments between the two architectures.

In comparison to the SPES methodology, CESAR shares many commonalities. Functional analysis, logical architecture and physical architecture can be found in SPES in the functional, logical and technical viewpoint. The requirements viewpoint in SPES can be described as a combination of operational analysis and with part of the functional/non-functional analysis, as requirements are described in both step (operational analysis, functional/non-functional analysis) in CESAR. This methodology does also not provide a dedicated methodology enabling exploration in order to automatically calculate E/E Architectures.

3.4.3. IBM Harmony

With the tool independent *Rational Integrated Systems/Embedded Software Development Process Harmony*, IBM provides a general Model-Driven Development systems development

process. Based on this process they created a SysML profile of this process for Model-Based Systems Engineering with Rational Rhapsody and Rational Harmony for Systems Engineering [18]. It aims to provide system engineers with an integrated system and software development process.

Figure 3.7 shows a high-level overview over the IBM Harmony process. It is based on the general V-Model and consists of two main parts: Harmony for Systems Engineering and Harmony for Embedded Real-Time Development. Harmony for Systems Engineering describes a top-down design flow starting with the requirements analysis, system functional analysis and design synthesis using models on each level. While (executable) uses-cases are predominantly used in the first two phases, the system architecture is created during design synthesis. The system architecture baseline marks the transition to the Harmony for Embedded Real-Time development, analyzing, designing and implementing the software followed by the different test phases starting from unit to system acceptance test on the right side of the V-Model.

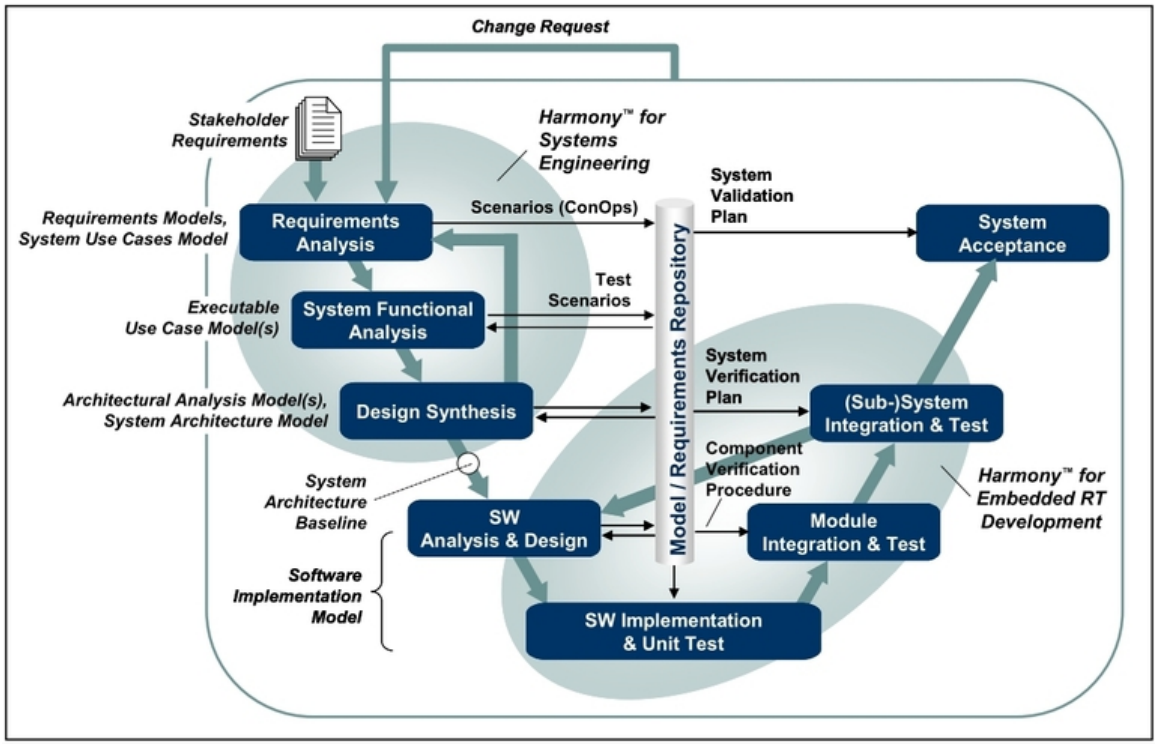


Figure 3.7.: Overview of the IBM Harmony process [18]

Although this process provides a seamless model driven development approach using models in all development phases, it only covers behavioral models until the software implementation models. The first three phases (requirements analysis, system functional analysis

and design synthesis) are similar to CESAR (operational analysis, functional/nf analysis, logical architecture) and SPES (requirements, functional and logical viewpoint). However, the modeling of technical/hardware aspects of the system is missing entirely. In CESAR, this is covered by the physical architecture and in SPES by the technical viewpoint.

3.4.4. AUTOSAR methodology

In addition to the AUTOSAR ADL (introduced in Section 3.1.3), the open industry standard also provides a methodology [19]. Figure 3.8 illustrates an overview over the general workflow which is proposed by the AUTOSAR methodology. The methodology thereby distinguishes between the development of Basic Software, System, Virtual Function Bus, Software Components, and the ECU executable displayed in dashed rectangles. Here, artifacts are displayed as envelopes (e.g. *System Constraint Description*) which are input and output of certain process steps (e.g. *Develop System*).

The AUTOSAR methodology is hence describing a workflow how one specific ECU can be implemented. The last development step describes how the software integrates into the ECU (*Integrate Software for ECU*). Compared to the other engineering methodologies - SPES, CESAR and IBM Harmony - the AUTOSAR methodology covers the low level development and implementation during system development, whereas the other methodologies propose a high level design of systems starting from requirements to functional, logical and technical design. Thereby, high-level development describes early design phases in the development, enabling to choose between different system designs. Low-level development describes the implementation of the system in terms of implementation of a concrete realization (e.g. coding). Due to this fact, in the AUTOSAR methodology, hardware and software are already fixed at this stage and there is no designated exploration of different possible architecture designs.

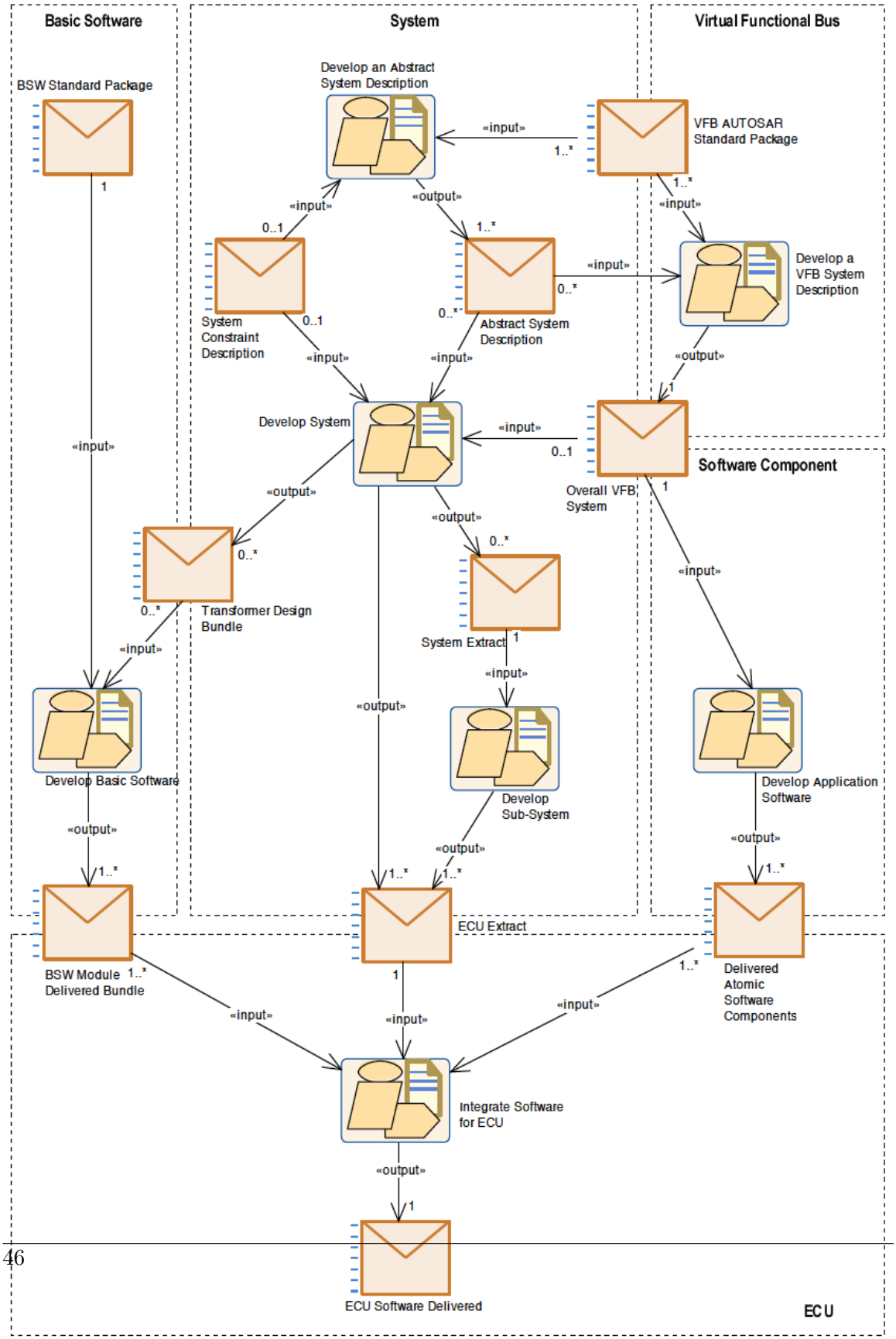


Figure 3.8.: General workflow of the AUTOSAR methodology [19]

4. E/E Architecture Exploration Methodology

In this Chapter, we describe a methodology to automatically explore the design space of E/E Architectures. Therefore, in Section 4.1, we describe an exploration process, its composition by means of viewpoints and the relations between those viewpoints. Building upon this process, will show how our approach can be integrated into a development process in Section 4.2. On the one hand, we describe how the proposed process and viewpoints integrate into a customized V-Model XT development approach. On the other hand, we demonstrate how our approach integrates into the existing system engineering methodology SPES.

The following Chapter has been published in Eder et al. [24] in a first version but has been largely adapted and rewritten in the course of this thesis.

4.1. Exploration Process

In order to be able to perform an exploration of E/E Architectures, we are structuring the exploration according to the concept of viewpoints (cf. Section 2.2.2). Each viewpoint describes different views on a specific exploration aspect. We identified three different Exploration Viewpoints:

1. **E/E Architecture Viewpoint**

   Describes dedicated meta-models in order to to describe an E/E Architecture covering software, hardware and deployment aspects.

2. **Specification Viewpoint**

   Describes a domain specific language, enabling the formalization of an E/E Architecture exploration problem.

3. **Exploration Viewpoint**

   Describes how an automatic exploration of E/E Architectures can be preformed.

In the following, we introduce each of the viewpoints and show their relations in terms of process steps.

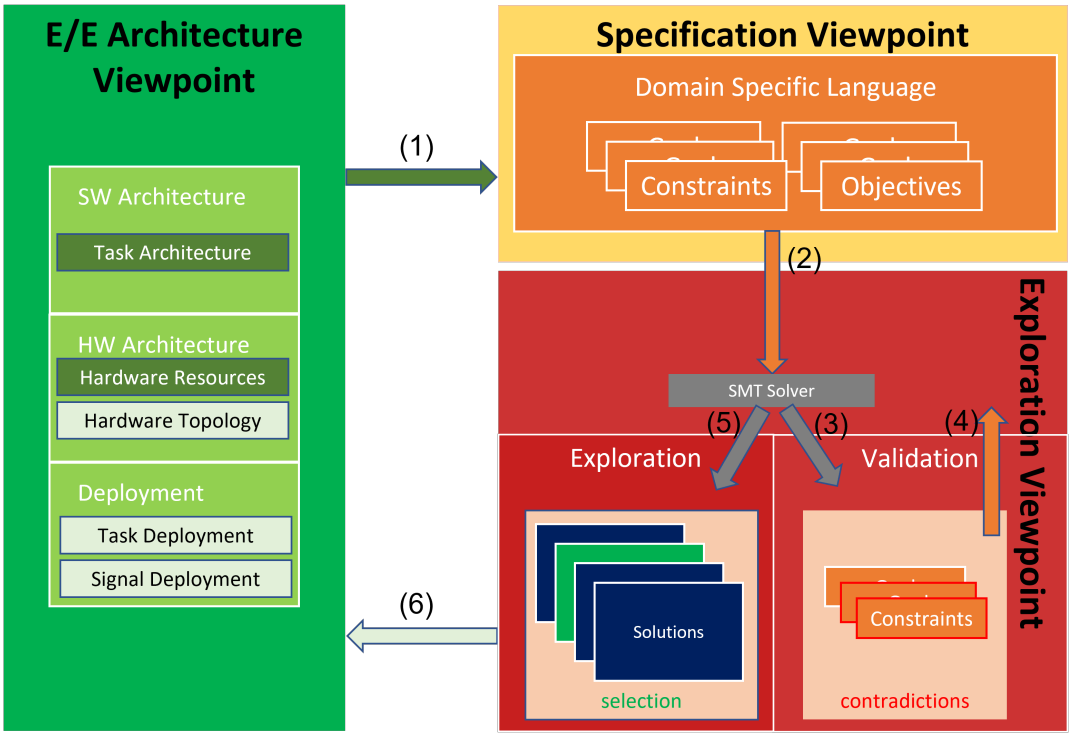


Figure 4.1.: Structure of the proposed viewpoints and their relations. The numbers in brackets (1)-(6) illustrate the different process steps and are described in more detail below.

4.1.1. E/E Architecture Viewpoint

The E/E Architecture Viewpoint describes all models which enable modeling an E/E Architecture. They are described in detail in chapter 5 and consist of the following models.

- **Software Architecture**

  Describes software models representing the the behavior of the system in terms of functionality.

- **Hardware Architecture**

  Describes hardware models in terms of, e.g., computation and communication models.

- **Deployment**

  Describes the connection between Software Architecture and Hardware Architecture.

The purpose of those models is to enable the design of E/E Architectures already at early stages of development. The process steps between the E/E Architecture Viewpoint, the Specification and the Exploration Viewpoint are as follows.

(1) In order to support an automatic exploration of E/E Architectures, part of the software and hardware models are used as an input for the Specification Viewpoint.

(6) After an exploration has been performed in the Exploration Viewpoint, the results, in form of hardware and deployment models, are fed back into the E/E Architecture Viewpoint.

4.1.2. Specification Viewpoint

The Specification Viewpoint provides means to formally describe an E/E Architecture exploration problem. This is achieved by a domain specific language (DSL) which is explained in more detail in Chapter 6. This viewpoint can be described by the following concepts.

- **Domain Specific Language**

  A DSL enables to abstract away from exploration specific implementations and enables using different exploration technologies on the basis of a single language which has to be transformed into the respective technology.

- **Constraints**

  On the one hand, the DSL enables the formalization of constraints. This allows the formalization of certain requirements which have to hold in order to explore a valid E/E Architectures.

- **Objectives**

  On the other hand, the DSL enables the formalization of objectives. This allows the formalization of optimization goals (which may also be derived from certain requirements) which optimize an E/E Architecture, e.g. considering cost.

The process steps of the Specification Viewpoint to the E/E Architecture Viewpoint and the Exploration are as follows:

(1) The E/E Architecture models which are received in the Specification Viewpoint, are transformed into language constructs. By that, an E/E Architecture exploration problem is formalized.

(2) After all constraints and objectives have been defined, this formalized E/E Architecture exploration problem is passed to the Exploration Viewpoint.

4.1.3. Exploration Viewpoint

The Exploration Viewpoint, provides the specific technologies which enable an automatic calculation of E/E Architectures. In this thesis, we are using the Z3 SMT solver in order to solve an E/E Architecture exploration problem. The viewpoint is described in detail in Chapter 7. In general, we distinguish between two steps, in this viewpoint.

- **Validation**

  The Validation step, checks for contradictions in the E/E Architecture exploration problem. If there are contradictions, there is no solution for the problem and thus no E/E Architecture can be generated. This means that the contradictions have to be resolved at first.

- **Exploration**

  In the Exploration step, the goal is to calculate different optimized E/E Architectures. As the optimization objectives in an E/E Architecture exploration are often inversely influencing each other, the results mostly depict trade-offs (e.g. low costs mean higher energy consumption, lower energy consumption mean higher costs).

The process steps within the Exploration Viewpoint and to the Specification and E/E Architecture Viewpoint are as follows.

(2) The constraints and objectives which describe the E/E Architecture exploration problem in the Specification Viewpoint, are transformed into SMT, in order to be solved by the Z3 SMT solver.

(3) Using this solver, the validation step is then checking for contradictions between the constraints.

(4) In case there are contradictions, the contradicting constraints are highlighted and have to be resolved. Such a contradiction may give a hint about contradicting requirements, as the constraints might be derived from the requirements of the system. After the contradictions have been resolved, step (2) has to be performed again.

(5) After a successful validation, multiple optimized E/E Architectures can be generated in the exploration step.

(6) After one E/E Architecture has been chosen, the solution is transformed into Hardware and Deployment models in the E/E Architecture Viewpoint.

4.2. Integration into the development process

In the following, we describe how the proposed exploration methodology integrates into system development. Hence, we show, on the one hand, how the the approach integrates

into the V-Modell XT development approach (Section 4.2.1) and, on the other hand, into the embedded system development methodology SPES (Section 4.2.2).

4.2.1. V-Modell XT

The V-Modell XT[1] is a de-facto standard for the development of software intensive embedded systems in Germany. It has been developed in order to react to a rising number of (economically) non-successful or failing projects developing software intensive systems [82]. The XT stands for extreme tailoring which is the main idea of the standard, namwley the provision of a framework of building blocks which can be tailored to the needs of specific projects. The V-Modell XT is a trademark of the Federal Republic of Germany.

It proposes several reference implementations like, e.g., *reference products*, *reference processes* and *reference roles*. In the following, we focus on the proposed *reference roles* and schematically define a development process, based on some of those *roles*. We then integrate our exploration methodology into such a development process and show that there is an additional role needed, in order to support an exploration.

4.2.1.1. Roles in V-Modell XT

The V-Modell XT proposes several *reference role*s in the development process. In the following, we have a closer look at a selected number of roles, that enable an integration of the proposed exploration methodology into such a development process.

- Requirements Analyst

  Creation, assessment and refinement of functional and non-functional requirements.

- System Architect

  Design of the System Architecture which entails the technical design, the tracing to the requirements, definition of interfaces and specification of HW/SW interfaces.

- Software Architect

  Design of the software architecture and realization of requirements and definition of software units. The software architect also contributes to the design of the system architecture.

- Software Developer

  Realization (programming) of software units and integration into the system.

- Hardware Architect

  Design of the Hardware Architecture and realization of requirements and definition of hardware units. Selection of mechanical or electronic components.

[1] `http://ftp.tu-clausthal.de/pub/institute/informatik/v-modell-xt/Releases/2.3/V-Modell-XT-Gesamt.pdf`

- Hardware Developer

  Realization of hardware units and integration into the system.

- System Integrator

  Integration of the different hardware and software units which form the overall system.

Figure 4.2 illustrates these roles aligned to the different design stages of a classical V-Model development approach: *requirements analysis*, *high-level design*, *low-level design*, *unit design* and *integration*. Despite the sequential order, the different stages are supposed to be executed in parallel, iteratively refining the system under development.

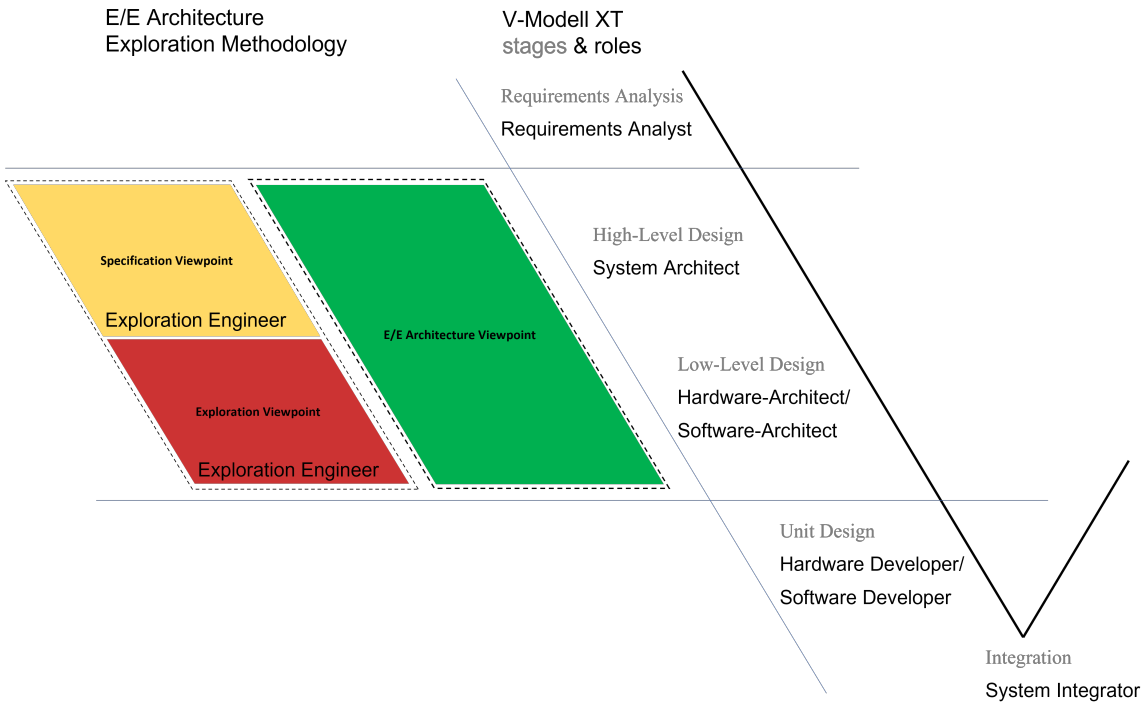


Figure 4.2.: Roles at different stages in the development process using a customized V-Modell XT. (Each development stage (written in gray) is accompanied by the role which is responsible in this stage (written in black below the stage).)

The E/E Architecture Viewpoint (highlighted green in Figure 4.2) is mainly used during High-Level and Low-Level (System) Design. Thus, it is used already at early stages of development, where a system architect is defining the system architecture. Considering an automotive E/E Architecture, this entails answering the following questions: how to design the system architecture of the system and how to distribute functionality within the system?

In order to answer these questions, the *system architect* needs to aggregate knowledge from the *requirements analysis* stage and from the *low-level design* stage. On the one hand, the *requirements analysis* might provide constraints and objectives for the system architecture such as compliance with a safety standard. On the other hand, the *low-level design* implicitly provides system constraints such as available memory of certain hardware.

As introduced in Section 4.1.1, the E/E Architecture Viewpoint distinguishes between software and hardware architecture models. Thus, the *software architect* and *hardware architect* are also involved in the creation of the system architecture. Consequently, regarding the roles which we just introduced, the *software architect* is responsible for the software architecture and the *hardware architect* for the hardware architecture. The *system architect* is responsible for the whole E/E Architecture Viewpoint, deciding especially about the design of the hardware architecture, according to the available hardware resources and according to the deployment of the software architecture onto the hardware architecture.

4.2.1.2. Exploration Engineer as a new role

In order to support the *system architect* in building the E/E Architecture, the Specification and Exploration Viewpoint provide means to automatically calculate a system architecture given the input of the different models provided in the E/E Architecture Viewpoint.

Those viewpoints are located in parallel to the E/E Architecture Viewpoint (highlighted in yellow and green in Figure 4.2) as they shall support the *system architect*'s work to come up with a system architecture already at early stages of the development. Hence, their purpose is to enable and support frontloading activities allowing to estimate, for instance the cost of a system architecture, as well as to find contradicting system configurations and/or requirements as early as possible.

Because of that, we propose the new role of an ***exploration engineer*** which supports a *system architect* already at early stages of development. The *exploration engineer* is responsible for the Specification and the Exploration Viewpoint. He/she is able to aggregate the information of the E/E Architecture Viewpoint which is provided by the *system architect* and is furthermore able to use and extend a dedicated domain specific language, capable of expressing all necessary properties of the system. His/her task is then to formalize the input of the E/E Architecture Viewpoint in terms of constraints and objectives, by using this domain specific language. This formalization is then used in the Exploration Viewpoint to automatically validate the constraints ensuring that there are no contradicting requirements, on the one hand. On the other hand, after a successful validation, the exploration automatically calculates different system architectures. Those solutions are then transformed back into the E/E Architecture Viewpoint and can thus be used by the *system architect*. By that, the *exploration engineer* can support the *system architect* in designing the system architecture.

4.2.2. Integration into the SPES methodology

The development methodology SPES (Software Platform Embedded Systems [16][81]) has been developed throughout two consecutive German research projects (see also Chapter 3.4.1). Figure 4.3 shows, that SPES divides system development into four viewpoints: the requirements, the functional, the logical and the technical viewpoint. The models in each viewpoint can be developed on different levels of granularity.

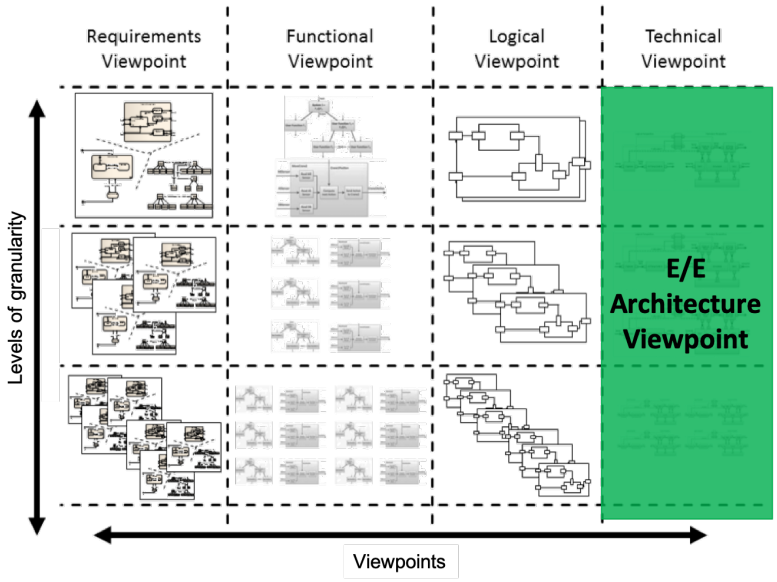


Figure 4.3.: Integration of the E/E Architecture Viewpoint into the technical viewpoint in SPES [16]

The methodology is especially targeted at developing embedded systems and can be described as a framework which is tailored to domain and project specific needs. In this work, we show how the proposed exploration methodology seamlessly integrates into the SPES framework and how it integrates both development methodologies into the bigger picture of the general V-Modell XT development approach.

Figure 4.4 exemplarily depicts the transition between requirements, functional, logical and technical viewpoint on one level of granularity. Starting from the requirements viewpoint describing, e.g., a use case model we are deriving the systems functionality from this diagram in the functional viewpoint. In the functional viewpoint, the functionality of the system is decomposed into two sub-functions. Those functions are the basis for the creation of the logical architecture in the logical viewpoint. This architecture is the first draft of the system architecture including all functional (through the deployment of functions to logical components) and non-functional aspects of the system but still platform independent. In the technical viewpoint, the implementation of the system is described in terms of software

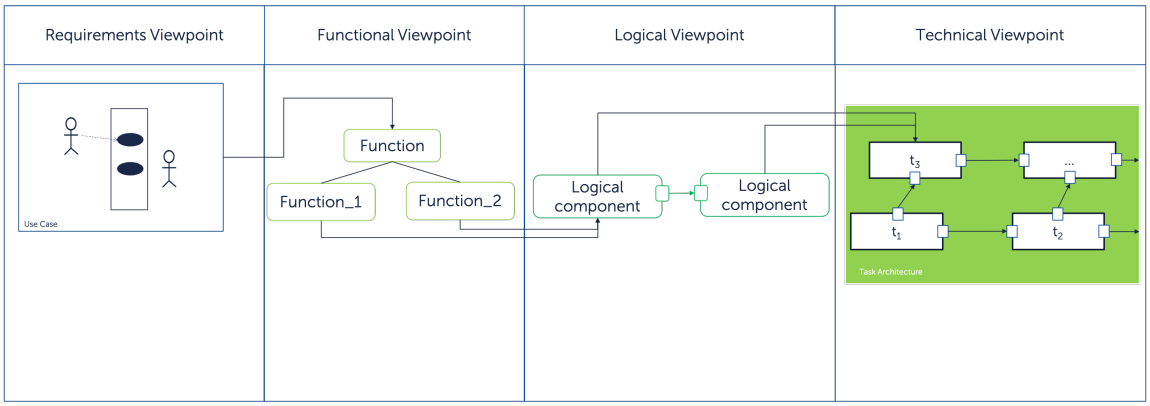


Figure 4.4.: Schematic transition from functional to logical to technical viewpoint

and hardware architectures and deployment. Hereby the logical components from the logical viewpoint are allocated to software tasks which realize the functionality of the logical component. However, SPES remains rather vague about the different meta-models and their dependencies in the technical viewpoint. Therefore, the proposed E/E Architecture Viewpoint in this thesis, provides dedicated technical meta-models (Chapter 5) and is thus an extension of the technical viewpoint of SPES. Additionally, with the Specification and Exploration Viewpoint, it provides a powerful extension to the technical viewpoint enabling the automatic generation of E/E Architectures.

Considering the development process according to the V-Model XT, depicted in 4.2.1, the SPES methodology seamlessly integrates into the tailored development process as illustrated in Figure 4.5. Therefore, we distribute the development *stages*, *roles* and *viewpoints* as follows:

- The *Requirements Viewpoint* can be mapped to the stage of *requirements analysis* which is part of the responsibility of the *requirements analyst*.
- The *Functional Viewpoint* is also part of *requirements analysis* as it structures the requirements as system functions. It is also part of the responsibility of the *requirements analyst*.
- The *Logical Viewpoint* describes the system architecture in terms of logical components but is still platform independent. It is created during *High-Level Design* and is part of the responsibility of the *system architect*.
- The *Technical Viewpoint* describes the technical realization of the system. It is part of the *High-Level Design* but also of the *Low-Level Design* as it considers the system architecture on the logical as well as on the software and hardware level. Consequently, it is part of the responsibility of the *system architect*, and the *software and hardware architect*.

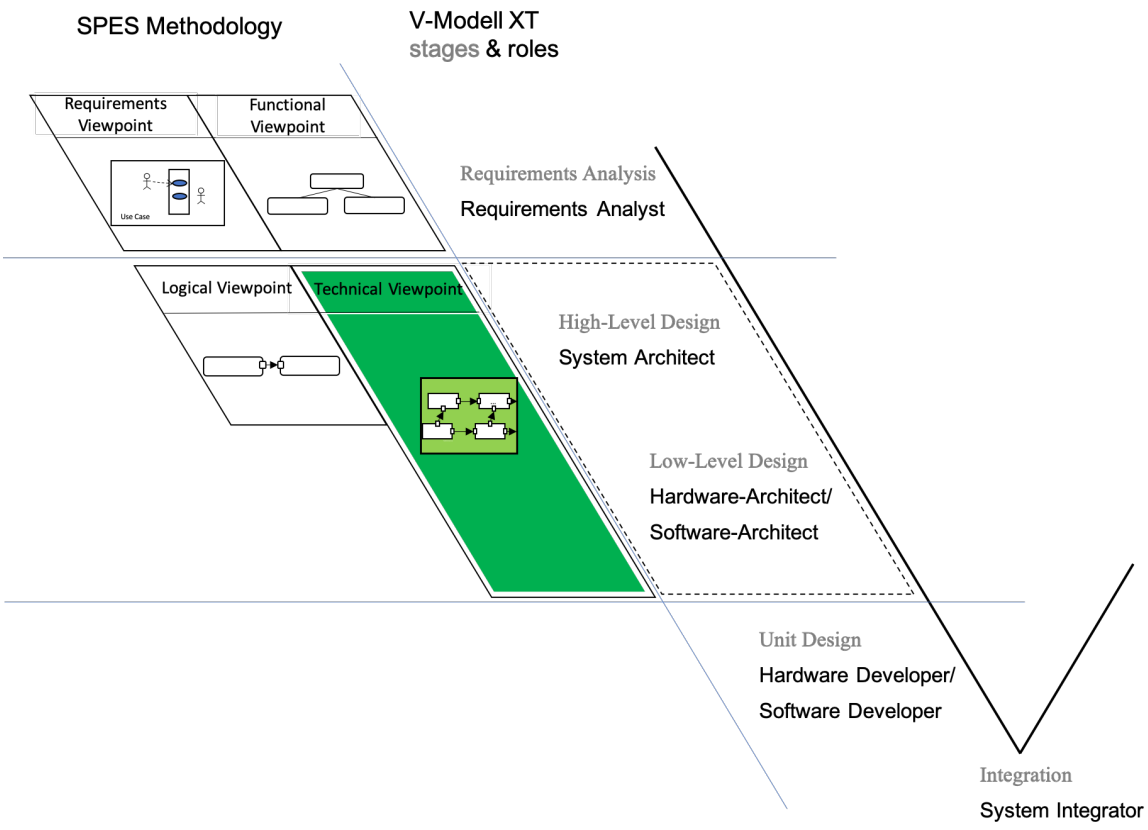


Figure 4.5.: Integration of the SPES Viewpoints and the proposed E/E Architecture Viewpoint (green) into the tailored V-Model XT development approach. (Each development stage (written in gray) is accompanied by the role which is responsible in this stage (written in black below the stage).)

5. E/E Architecture Viewpoint

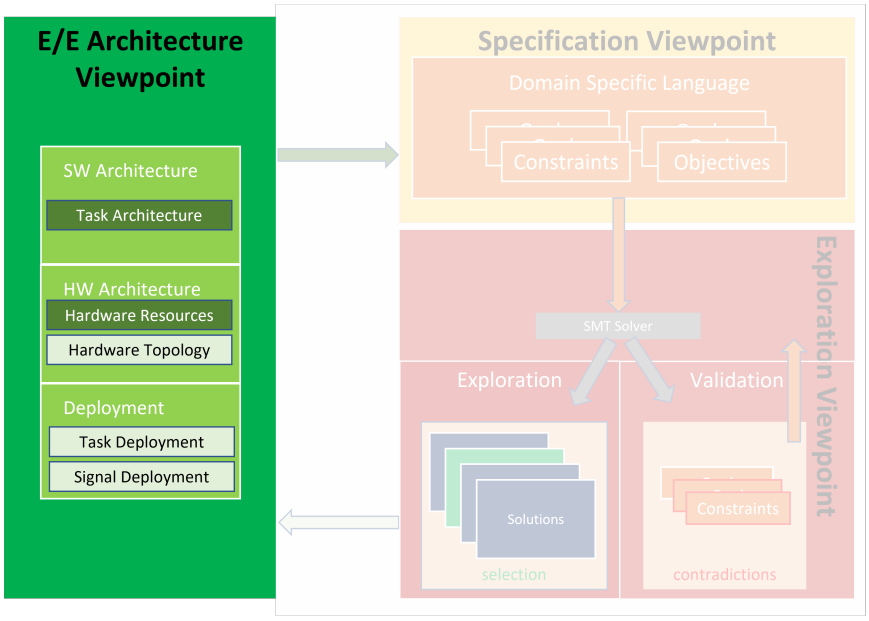


Figure 5.1.: Overview of viewpoints

The E/E Architecture Viewpoint describes the structure and the meta-models in order to describe an E/E Architecture. In this viewpoint, the system is described in terms of its technical aspects. We therefore distinguish between three different types of models:

1. Software Architecture
   a) Run-Time Software
   b) Design-Time Software
2. Hardware Architecture
   a) Hardware Resources
   b) Hardware Topology
3. Deployment

a) Task Deployment

b) Signal Deployment

1 The Software Architecture can be divided into a dynamic part called run-time software (1a) and a static part called design time software (1b). The run-time software is described through a Task Architecture and a schedule of *Task*s and messages. The design-time software includes the description of the operating system, middleware and virtualization aspects like for example partitioning. In this thesis, we will focus only on the run-time software, in particular on the Task Architecture. Thus the scheduling aspects as well as the design-time software is only mentioned for the sake of completeness here, and will not be described in detail in the following.

2 The Hardware Architecture describes the *Hardware Resources* (2a) and the *Hardware Topology* (2b). The *Hardware Resources* describe a set of computation resources (*ProcessingUnit*s) and communications resources (*Bus*es). The specific properties of those resources are described as annotations such as safety level or memory while Connectors attached to the resources describe possible interfaces between processing and communication resources. Furthermore, variability of those resources is described if more than one variant of a specific resource exists. Variants of a specific resource differ in their properties. The *Hardware Topology* describes the connections between computation and communication resources according to the interfaces described by Connectors.

3 The Deployment is connecting the Software Architecture and the Hardware Architecture by deploying the Task Architecture onto the *Hardware Topology*. Specifically, it describes the *Task* to *ProcessingUnit* deployment (3a), which defines where a *Task* is executed at run time and it describes the *Signal* to *Bus* deployment (3b) which determines the communication resource that is used to send this *Signal*.

An automotive E/E Architecture can thus be described by a Task Architecture, a *Hardware Topology* and a Deployment of the Task Architecture onto the *Hardware Topology*.

In the following, we describe each of the previously mentioned models in detail by means of their meta-models accompanied by an example. The Software Architecture will be explained in Section 5.1, the Hardware Architecture in Section 5.2 and the Deployment in Section 5.3.

This Chapter has been published in Eder et al. [24] in a first version and has been taken over and adapted for this thesis.

5.1. Software Architecture

The dynamic part of the Software Architecture is the run-time software. It is composed of a Task Architecture consisting of *Task*s which communicate via *Signal*s and a schedule of *Task*s and *Signal*s. It is called run-time software due to the fact that *Task*s, *Signal*s and schedules are executed/sent at run-time. Thus it describes the dynamic part of the Software Architecture.

Considering an E/E Architecture, the Software Architecture describes the software as *Task*s - which can be executed on a *ProcessingUnit* (ECU) - and *Signal*s exchanged between *Task*s - which can be sent via *Bus*es.

5.1.1. Meta-Model

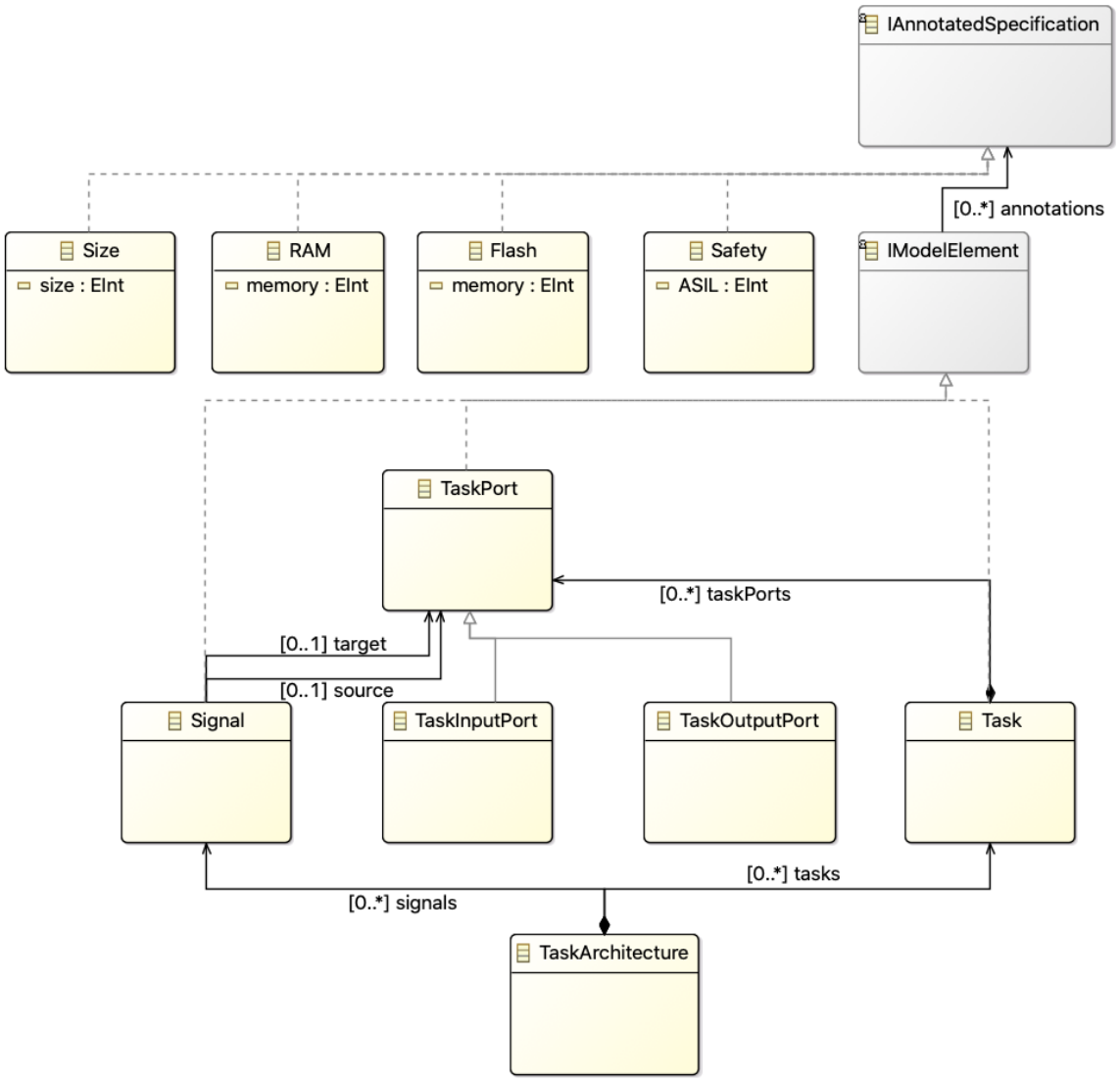


Figure 5.2.: Meta-Model of the Task Architecture

Figure 5.2 shows the meta-model of the Task Architecture. A *TaskArchitecture* consists of 0..* *Task*s and 0..* *Signal*s. Each *Task* may contain 0..* *TaskPort*s. A *TaskPort* can

either be of type *TaskOutputPort* or *TaskInputPort*. A *TaskOutputPort* is the source of a *Signal* sent between two *Task*s. A *TaskInputPort* is the target of a *Signal*. Thus the Task Architecture is described in a typical component-based way like e.g. proposed in [34].

Task, *TaskPort* and *Signal* inherit from the abstract interface *IModelElement*. Therefore, we also refer to them as model elements. *IModelElement* is a generalized concept to describe elements which can be created and/or altered e.g. by a user. At this level the concept of I*AnnotatedSpecification*s (also referred to as annotation in the following) is implemented. Any model element may contain an arbitrary number of IAnnotatedSpecifications. Figure 5.2 depicts exemplary annotations for elements of the Task Architecture. Specifically, the model elements of the Task Architecture contain the following annotations:

1. *Task*
   - RAM [Byte] (memory : int) describes the amount of Random Access Memory which is required to execute the *Task*
   - Flash [Byte] (flash : int) describes the amount of Flash memory which required to store the execution specification (e.g. Code) and execute the *Task*.
   - Safety [ASIL] (asil : ASIL) describes the automotive safety level ASIL of this *Task*
2. *TaskOutputPort*
   - Size [Byte] describes the size of the *Signal* which is sent by this *TaskOutputPort*

5.1.2. Example

Figure 5.3 shows an instantiated Task Architecture. It consists of Task_1 which has one *TaskOutputPort* atttached and Task_2 which has one *TaskInputPort* attached. Through those *TaskPort*s Task_1 communicates with Task_2 via a *Signal* whose source is the *TaskOutputPort* at Task_1 and whose source is the *TaskInputPort* at Task_2. Moreover, using the concept of annotations we can describe that Task_1 requires 64kByte of RAM and has an Automotive Safety Integrity Level (ASIL) A. The size of the *Signal* which is produced by Task_1 is 16 kByte. Task_2 requires 128kByte of RAM and an ASIL D.

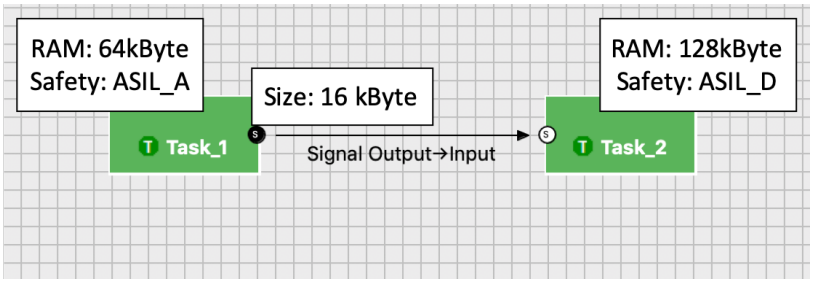


Figure 5.3.: Exemplary Task Architecture with two *Task*s, two *TaskPort*s and one *Signal*

5.2. Hardware Architecture

The Hardware Architecture describes the *Hardware Resources* and their *Hardware Topology*. The Hardware Architecture also covers variability of resources.

Considering an E/E Architecture, the Hardware Architecture describes *ProcessingUnit*s (ECUs) and *Bus*es (Hardware Resources) which are connected with each other to enable a communication between *ProcessingUnit*s via *Bus*es (Hardware Topology).

5.2.1. Hardware Resources

The *Hardware Resources* describe a 150% model [83, 84] of all *ProcessingUnit*s and *Bus*es, and their respective interfaces. In particular, this means that all variants of a specific Hardware Resource are described. Additionally, different types of resources are defined here, too.

5.2.1.1. Meta-Model

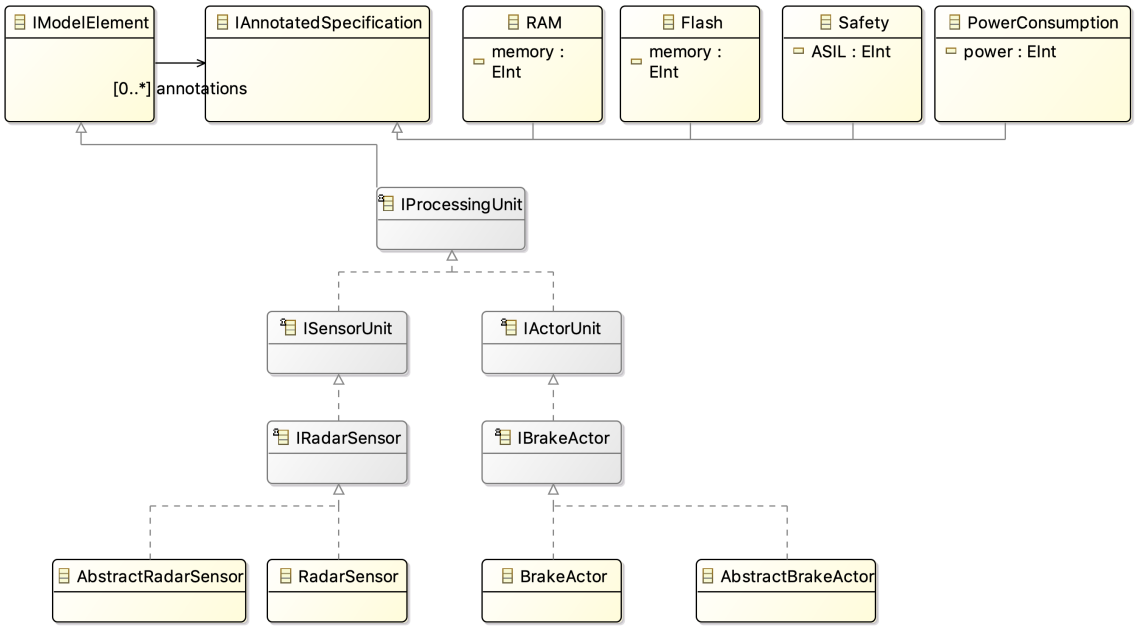


Figure 5.4.: Hardware Resource types

Figure 5.4 exemplarily shows the different types of *ProcessingUnit*s and *Bus*es. In this meta-model, both elements are described as an interface. Additionally, this meta-model is extended by an interface layer which details the types of *ProcessingUnit*s. A *ProcessingUnit* may thus be e.g. of type *ISensor* and more specifically of type *IRadarSensor*. For reasons of

clarity and comprehensibility we are only showing part of the whole technical meta-model here. A *ProcessingUnit* can furthermore be of type *IActuator* or of a different type of sensor like *ICameraSensor* to mention just a few.

In order to be able to express variability aspects in the E/E Architecture Viewpoint, we introduced the concept of a variation point [83]. As proposed in [85], we integrated this concept directly into the meta-model.

Figure 5.5 illustrates, how variability is integrated into the meta-model of the *Hardware Resources*. The interface *IAlternative* depicts the different variants of a specific Hardware Resource. The base class *AlternativeVariationPointBase* contains an arbitrary number of *IAlternatives*. An *AbstractComponent* is thus a container for variants of either *IProcessingUnits* or *IBuses*. An *AlternativeBase* is a base class for any *IAlternative*.

Hence, a specific *IProcessingUnit* or *IBus* can be expressed as an abstract element. This entails that e.g. a CAN Bus can be expressed as a Variation Point through an *AbstractBus* element. This abstract element in turn can contain alternative variants (like e.g. CAN FD (Flexible Data-Rate), CAN HI (Highspeed) or CAN LO (Lowspeed). This means that the abstract element represents three different variants of a CAN Bus which in this case differ e.g. in their bandwidth value.

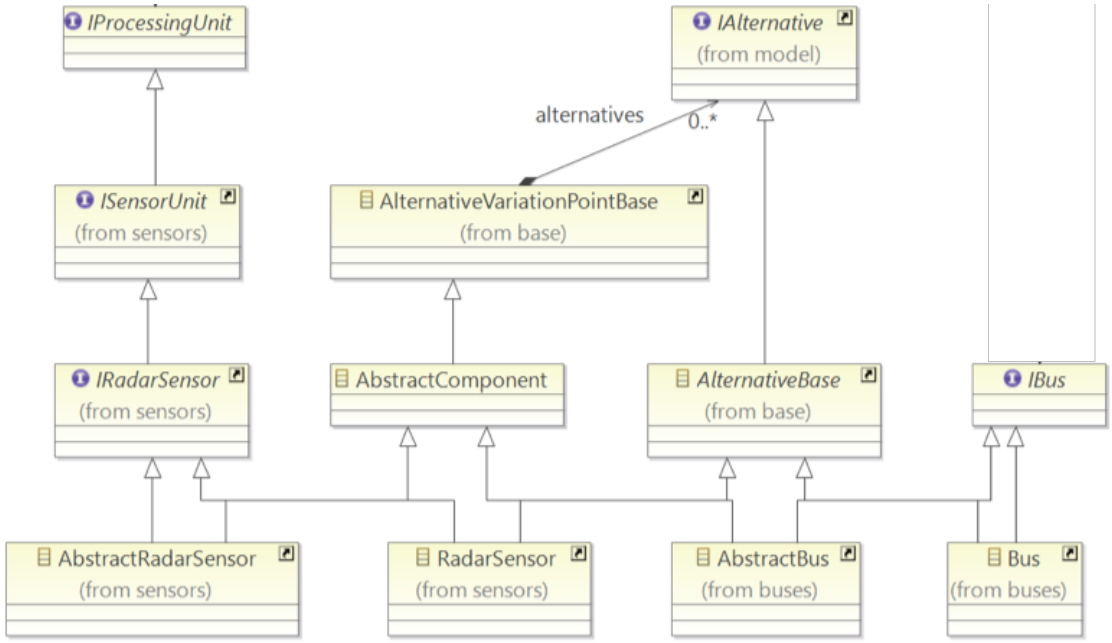


Figure 5.5.: Variability in the Hardware Architecture

Specifically, the model elements of the *Hardware Resources* contain the following annotations:

1. *ProcessingUnit*

- RAM [Byte] (memory : int) describes the amount of Random Access Memory which is required to execute the *Task*
- Flash [Byte] (flash : int) describes the amount of Flash memory which required to store the execution specification (e.g. Code) and execute the *Task*.
- Safety [ASIL] (asil : int) describes the automotive safety level ASIL of this *ProcessingUnit*
- Cost [€] or [$] (cost : int) describes the cost of this *ProcessingUnit*
- Power Consumption [W] (power : int) describes the power consumption of this *ProcessingUnit* in watts

2. *Bus*
   - Bandwidth [Byte/ms] (bw : int) describes the maximum bandwidth of the referring *Bus*

5.2.1.2. Example

Figure 5.6 shows an exemplary set of *Hardware Resources*.

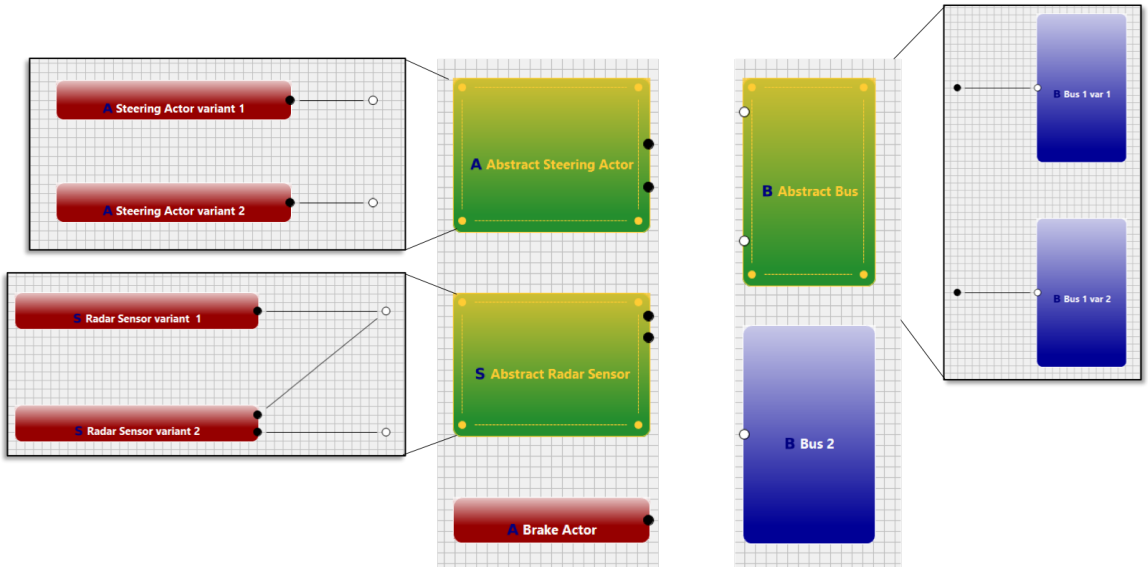


Figure 5.6.: Exemplary set of *Hardware Resources*. Green Elements depict *AbstractComponents* which contain different variants of this resource (*Abstract Bus* contains two variants, *Abstract Steering Actor* and *Abstract Radar Sensor* also contain two variants each). Non-green resources (*Brake Actor* and *Bus 2*) depict concrete resources for which no variants exist. The Connectors are shown as black (*IProcessingUnit*) or white (*IBus*) circles.

5.2.2. Hardware Topology

The *Hardware Topology*, as opposed to the *Hardware Resources*, describes a 100% model of all *Hardware Resources*. This means that, based on the interfaces of the *Hardware Resources*, a fixed connection between the different *Hardware Resources* is described.

5.2.2.1. Meta-Model

Figure 5.7 depicts the general meta-model of the *Hardware Topology*. A *Hardware Topology* consists of an arbitrary number of *IPhysicalPlatformArchitectureElements* which can be of type *IProcessingUnit*, *IBus* or *IPowerSupply*. A *IProcessingUnit* is able to execute Software. *Connector*s may be attached to it in order to establish a *BusConnection* to the Connectors of *IBus*es. A *IPowerSupply* maybe connected to I*ProcessingUnit*s via a *PowerConnection* between two *PowerConnector*s (one attached to each element) in order to supply an *IProcessingUnit* with electric power.

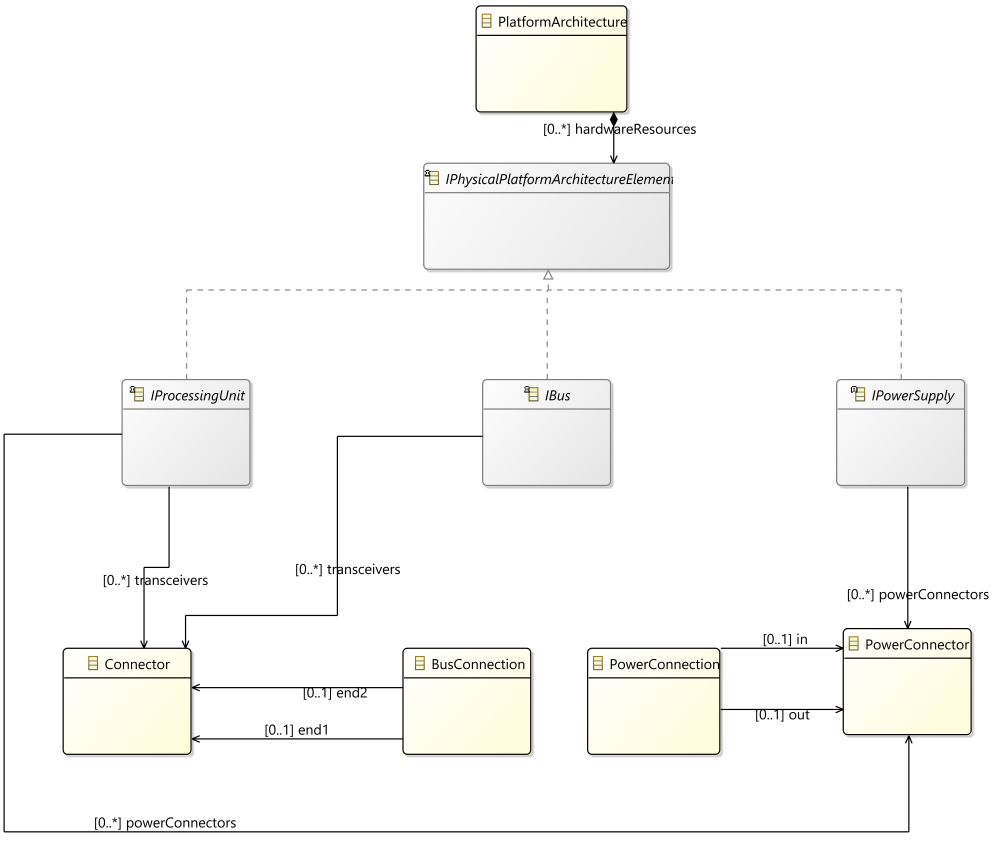


Figure 5.7.: Meta-model of the *Hardware Topology*

5.2.2.2. Example

Figure 5.8 shows an exemplary *Hardware Topology*.

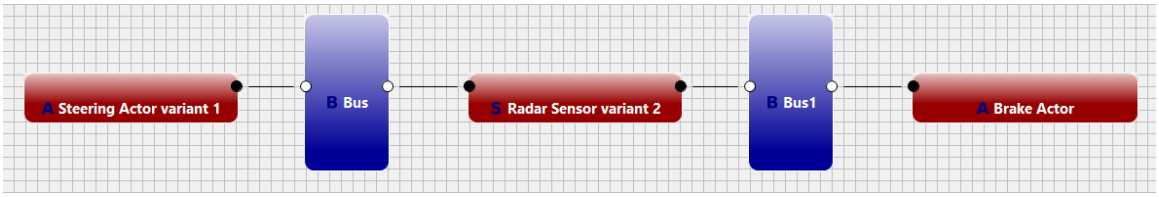


Figure 5.8.: Exemplary *Hardware Topology*. Concrete *IProcessingUnit*s (red) are connected to concrete *IBus*es (blue) via their Connectors (black and white circles). A *Hardware Topology* does not contain any *AbstractComponent*s. It is depicting a concrete *Hardware Topology*.

5.3. Deployment

The Deployment describes the connection between the Software Architecture and the Hardware Architectures. Specifically, all the *Task*s of the Task Architecture are allocated to a *ProcessingUnit* in the *Hardware Topology*, which entails that a *Task* will be executed on the *ProcessingUnit* it is allocated to. Similarly, the *Signal*s are allocated to *Bus*es, which means that a *Signal* will be sent via the *Bus* it is allocated to. Hence, a Deployment is defined as a set of *Allocation*s, where an *Allocation* defines the assignment of one *Task*/*Signal* to one *ProcessingUnit*/*Bus*.

The Deployment completes the description of an E/E Architecture as it connects the Software Architecture - described as a Task Architecture - and the Hardware Architecture - described as a *Hardware Topology* of *Hardware Resources*.

5.3.1. Meta-Model

Figure 5.9 shows the deployment meta-model. A Deployment consists of an arbitrary number of *Allocation*s. An *Allocation* is described using generic types. An *Allocation* source S must of type *IModelElement* and a target T also of type *IModelElement*. Hence, we can derive two specific Deployment, *TaskDeployment* and *SignalDeployment*. A *TaskDeployment* consists of *Allocation*s from *Task* to *ProcessingUnit*. A *SignalDeployment* consists of *Allocation*s from *Signal* to *Bus*.

5.3.2. Example

Figure 5.10 depicts an exemplary Deployment. It shows how Software and Hardware Architecture are combined by deploying the *Task*s and *Signal*s of the Task Architecture onto a *Hardware Topology*.

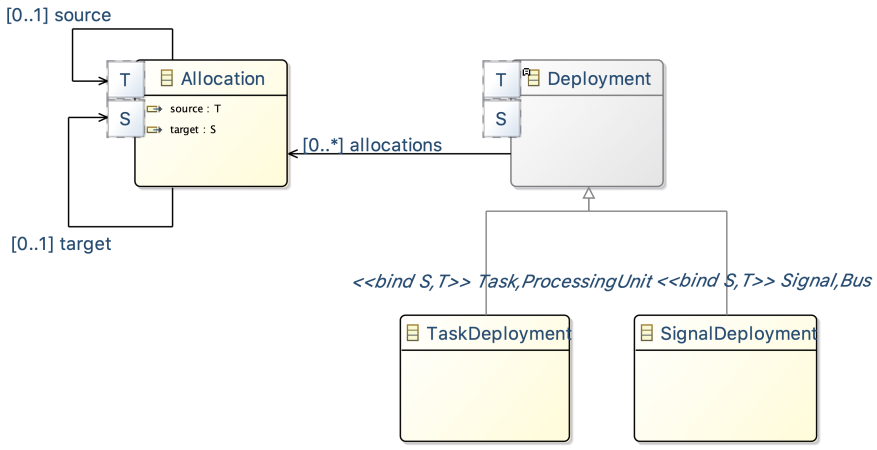


Figure 5.9.: Meta-model of the Deployment

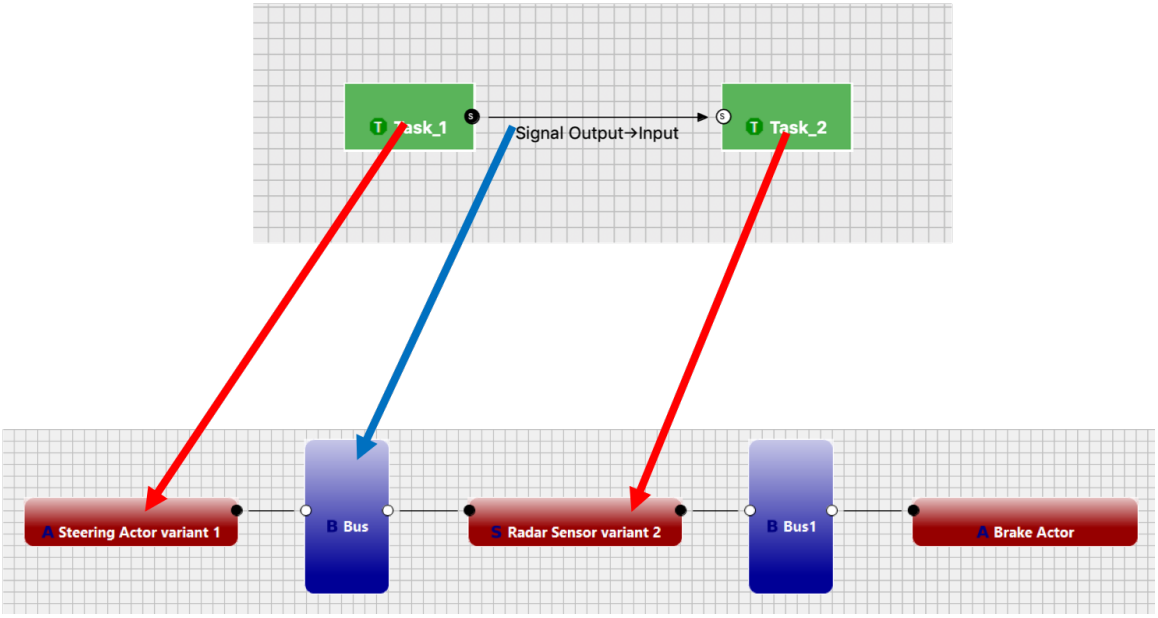


Figure 5.10.: Exemplary Deployment. *Task_1* is allocated to *Steering Actor variant 1* and *Task_2* is allocated to *Radar Sensor variant 2*. Consequently, the Signal which is sent between *Task_1* and *Task_2* is allocated to *Bus 1*, as both *IProcessingUnits* are connected to that bus via their *Connectors*.

6. Specification Viewpoint

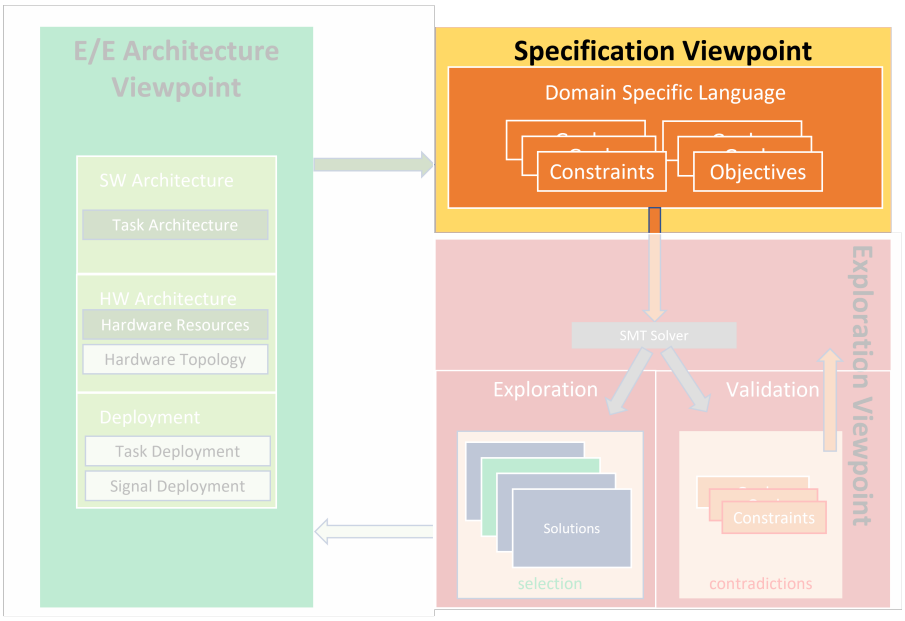


Figure 6.1.: Overview of viewpoints

The Specification Viewpoint provides means in order to formally describe an E/E Architecture Exploration problem. Therefore, we provide a domain specific modeling language (DSML) in order to define such a problem (cf. Section 6.1). The DSML provides an abstraction between the models described in the E/E Architecture Viewpoint and the definition of an E/E Architecture Exploration problem in the Exploration Viewpoint. Furthermore, we provide a set of language patterns to support re-usability (cf. Section 6.2). These patterns are commonly used for exploration tasks when engineering E/E Architectures.

This Chapter has been published in Eder et al. [24] in a first version and has been taken over and adapted for this thesis.

Table 6.1.: DSML Grammar

<DSML-Formula>	::=	<Constraint>| <Objective>
<Objective>	::=	**min** <ArithExpr>| **max** <ArithExpr>
<Constraint>	::=	<BooleanExpr>
<ArithExpr>	::=	<ArithExpr><ArithOperator><ArithExpr>| <AggrFunExpr>| <AtomicArithExpr>| <ModeElemPropertyFun>
<BooleanExpr>	::=	<BooleanExpr><BooleanOperator><BooleanExpr>| <ArithExpr><CmpOperator><ArithExpr>| <ModelElement><EquOperator><ModelElement>| **NOT** <BooleanExpr>| <QuantifierExpr>| <AtomicBoolExpr>
<QuantifierExpr>	::=	**forAll** <ModelElementSet><BooleanExpr>| **exists** <ModelElementSet><BooleanExpr>
<BooleanOperator>	::=	**AND** | **OR**
<CmpOperator>	::=	< | > | <= | >= | <EquOperator>
<EquOperator>	::=	== | **!=**
<ArithOperator>	::=	+ | − | * | /
<ModelElementSet>	::=	<ModelElement>*
<AtomicBoolExpr>	::=	**True** | **False**
<AtomicArithExpr>	::=	**Integer** | **Double**
<AggrFunExpr>	::=	**Sum** <ModelElementSet><BoolExpr><ArithExpr>| **Count** <ModelElementSet>
<ModelElement>	::=	**Task** | **Signal** | **ProcessingUnit** | **Bus** | **Connector** | <ModelElementFunExpr>
<ModElemProperty>	::=	**SafetyLevel** | **RAM** | **Flash** | **Power** | **Cost** | **Size** | **Bandwidth**
<ModElemPropertyFun>	::=	**property** <ModelElement><ModElemProperty>
<ModelElementFunExpr>	::=	**start** <ModelElement>| **end** <ModelElement>| **connected** <ModelElement>| **allocate** <ModelElement>

6.1. A language for technical architecture exploration

The domain specific modeling language (DSML) is used to define constraints and objectives over model elements and their properties. Constraints are restricting the set of possible solutions and objectives are optimizing a solution into a certain direction. Both constraints and objectives can be formalized forms of system requirements. A simplified version of the grammar is presented in Table 6.1.

A *DSML-Formula* is either a *Constraint* or an *Objective*. A *Constraint* is a *BooleanExpr* which basically correspond to first-order logic with a set of predefined model element types and functions evaluating to either true or false. An *Objective* corresponds to a minimization or a maximization of an *ArithExpr*.

6.1.1. Boolean expressions

A *BooleanExpr* can be

- concatenation of two *BooleanExpr* with a binary *BooleanOperator AND* or *OR*

<BooleanExpr><BooleanOperator><BooleanExpr>,

- concatenation of two *ArithExpr* with a binary *CmpOperator* or *EquOperator* <, >, <=, >=, ==, ≠
  (<ArithExpr><CmpOperator><ArithExpr>| <ArithExpr><EquOperator>
  <ArithExpr>),

- concatenation of two *ModelElement* with a binary *EquOperator* ==, ≠
  (<ModelElement><EquOperator><ModelElement>)

- negation of a *BooleanExpr*
  (NOT <BooleanExpr>),

- an *AtomicBoolExpr* evaluating either to true or false
  ($TRUE$ or $FALSE$),

- a *QuantifierExpr* which a allows quantification over a set of elements.

6.1.2. Quantifier expressions

A *QuantifierExpr* can either be a

- universal quantifier *forAll* iterating over all *ModelElement*s of a *ModelElementSet* and demanding that a given *BoolExpr* has to hold for all of these elements
  ($\forall ModelElement \in ModelElementSet.BoolExpr$) or a

- existantial quantifier *exists* iterating over all *ModelElement*s of a *ModelElementSet* and demanding that a given *BoolExpr* has to hold for at least one of these elements
  ($\exists ModelElement \in ModelElementSet.BoolExpr$)

6.1.3. Arithmetic expressions

An *ArithExpr* can be a

- concatenation of two *ArithExpr* with a binary *ArithOperator* +, –, *, /
  (<ArithExpr ><ArithOperator><ArithExpr>),

- an *AtomicBoolExpr* evaluating either to an *Integer* or *Double* value,

- a *AggrFunExpr* which is an aggregating function which evaluates to an *ArithExpr*,

- a *ModelElemPropertyExpr* which evaluates an arithmetic property of a *ModelElement* an thus evaluates to an *ArithExpr*.

6.1.4. Aggregation function expressions

An *AggrFunExpr* can be a

- a *Sum* over the *ModelElement*s out of a given *ModelElementSet* for which a given predicate *BoolExpr* holds and which sums up the values of a certain *ArithExpr* for each *ModelElement*
  $((\sum_{\{ModelElement \in ModelElementSet | BoolExpr\}} ArithExpr))$
- a *Count* function which calculates the cardinality of a *ModelElementSet*
  $(|ModelElementSet|)$.

6.1.5. Model element function expressions

A *ModelElemFunExpr* can be a

- *start* function which evaluates the start of a given *ModelElement* by returning the respective *ModelElement*
  (start : ModelElement → ModelElement)
- *end* function which evaluates the end of a given *ModelElement* by returning the respective *ModelElement*
  (end : ModelElement → ModelElement)
- *connected* function which evaluates the connection of a certain *ModelElement* by returning the *ModelElement* it is connected to
  $(connected : ModelElement \rightarrow ModelElement)$
- *allocate* function which evaluates the *Allocation* of a certain *ModelElement* by returning the *ModelElement* it is allocated to
  $(allocate : ModelElement \rightarrow ModelElement)$

6.1.6. Model element expressions

Expressions related to *ModelElement*s can be

- a *ModelElement* corresponds to a *Task*, *Signal*, *ProcessingUnit*, *Bus* or *Connector*. The elements correspond to the models introduced in 5.1 and 5.2.
- a *ModelElementSet* consists out of a set of *ModelElement*s
  $(\{ModelElement_1, ModelElement_2, ..., ModelElement_n\})$,
- a *ModElemPropertyFun* which evaluates the property (*ModElemProperty*) of a certain *ModelElement*. The *ModElemProperty* are equal to the annotations of the models introduced in 5.1 and 5.2 and can be of type:
  - SafetyLevel

  - RAM
  - Flash
  - Power
  - Cost
  - Size
  - Bandwidth

  $(property : ModelElement \times ModElemProperty \rightarrow ArithExpr)$

6.2. Language Patterns

Requirements in system engineering serve as an input to this approach as they correspond to constraints and objectives. However, those requirements need to be formalized in order to be used as an input. Such a formalization of requirements to constraints and optimization objectives is a task that requires a certain know-how in formal methods. From our experience, practitioners are not always willing to deal with the full power of formal methods and languages. A way to simplify this process is to restrict it by providing patterns.

We distinguish between Basic Patterns (Section 6.2.1) which are inherent to any E/E Architecture Exploration problem and Constraint (Section 6.2.2) and Objective Patterns (Section 6.2.3)) which can be individually defined.

In the following formulas (and unless denoted otherwise), we will use the following abbreviations for *ModelElementSet*s:

- **T** refers to the set of all *Task*s (part of the Software Architecture introduced in Chapter 5.1).
- **S** refers to the set of all *Signal*s part of the Software Architecture introduced in Chapter 5.1).
- **B** refers to the set of all *Bus*es (part of the Hardware Architecture introduced in Chapter 5.2).
- **P** refers to the set of all *ProcessingUnit*s (part of the Hardware Architecture introduced in Chapter 5.2).
- **C** refers to the set of all *Connector*s (part of the Hardware Architecture introduced in Chapter 5.2).

6.2.1. Basic Patterns

The Basic Patterns describe Constraint Patterns which are necessary in order to describe a correct E/E Architecture. Thus, they are inherent to any formalization of an E/E Architecture Exploration problem. There exist two kinds of Basic Patterns which will be described in the following: the Topology Pattern and the Variability Pattern.

6.2.1.1. Topology Pattern

The Topology Pattern describes a specific constraint which is necessary in order to set up a valid Deployment from *Task*s to *ProcessingUnit*s as well as a valid Hardware Topology of *ProcessingUnit*s and *connected Bus*es.

The *connected* function is defined to take a *ModelElement* as input and also return a *ModelElement*. In order to define valid routes between *ProcessingUnit*s and *Bus*es we will define two *connected* functions:

$$connectedP : Connector \rightarrow ProcessingUnit \tag{6.1}$$
$$connectedB : Connector \rightarrow Bus \tag{6.2}$$

Function 6.1 defines the connection of a *ProcessingUnit* to a certain Connector. Similarly function definition 6.2 defines the connection of a *Bus* to a certain connector.

Together, those functions can then be combined to describe valid communication routes between *ProcessingUnit*s and *Bus*es. In particular, that means that a route between a *ProcessingUnit* p and a *Bus* b exists only if they have the same *Connector* c attached to them.We therefore define an intermediate helper function *route*, which enables the definition of the Hardware Topology, as follows:

$$route(p, b) := \exists c \in C.connectedP(c) = p \wedge connectedB(c) = b \tag{6.3}$$

Moreover, the *Connector* element allows defining typed connections between *ProcessingUnit*s and *Bus*es such that a specific *ProcessingUnit* variant may only be connected to a specific *Bus* variant. Figure 6.2 schematically illustrates such a usage of *Connector*s. The *Connector*s (red shapes) can be attached to variants (as introduced in the meta-model in Figure 5.7) of a certain abstract *ProcessingUnit* or abstract *Bus* via the *connected* function. Definition 6.3 enables to define all possible routes between the elements (dashed lines in fig. 6.2).

Having introduced the definition of routes between *ProcessingUnit*s and *Bus*es, they have to be linked to the deployment of *Task*s and *Signal*s. Definition (6.4-6.6) describes the Topology Pattern which is necessary to create a correct E/E Architecture. Here, the *Allocation* (of Tasks and Signals) is brought together with the Hardware Topology (defined by using Definition 6.3) by connecting the *Signal*s of the Software Architecture via the *start* and *end* function to the *Allocation* of *Task*s to *ProcessingUnit*s.

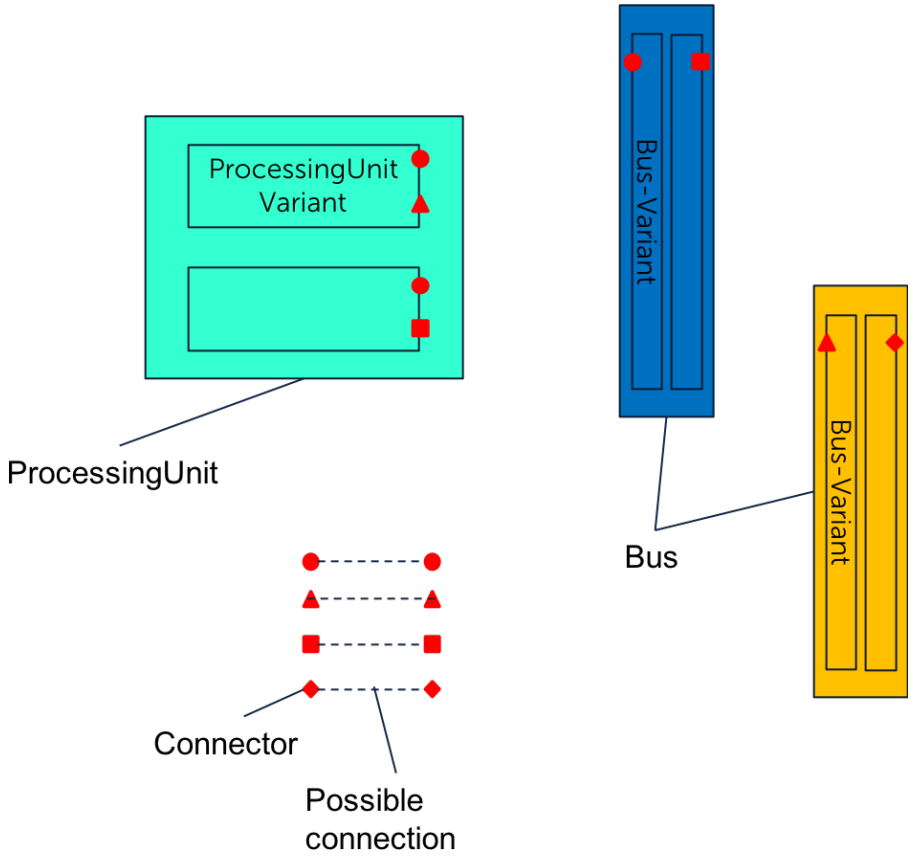


Figure 6.2.: Schematic usage of *Connector*s and corresponding routes in a Hardware Topology

$$\forall s \in S. \exists b \in B. \forall\ p_1, p_2 \in P.$$

$$allocate(start(s), p_1) \wedge allocate(end(s), p_2) \tag{6.4}$$

$$\Rightarrow$$

$$((p_1 \neq p_2) \wedge route(p_1, b) \wedge route(p_2, b) \wedge allocate(s) = b) \vee \tag{6.5}$$

$$((p_1 = p_2) \wedge allocate(s) = b_{null}) \tag{6.6}$$

The Topology Pattern (Definition 6.4-6.6) entails, that two *Task*s communicating with each other through a *Signal* s (implicitly $start(s) = task_1$ and $end(s) = task_2$)) may only be allocated to two *ProcessingUnit*s p_1, p_2 (line 6.4) if there is a route existing between

those *ProcessingUnit*s. In this case, the *Signal* s has to be allocated to the *Bus* b which is *connected* to both p_1, p_2 (line 6.5). In order to avoid unnecessary *Signal* to *Bus Allocation*s, s is allocated to a so called "null bus" (b_{null}) if both start and end *Task* of *Signal* s are allocated to the same *ProcessingUnit* (line 6.6). Note: the "null bus" is only temporarily created and removed afterwards.

6.2.1.2. Variability Pattern

The variability of Hardware Resources expressed through variants of *ProcessingUnit*s and *Bus*es as introduced in the meta-model in 5.5 is described by the Variability Pattern.

Expression (6.7) describes the variability constraint for a Hardware Resource (ProcessingUnit or Bus). Here, the set VAR denotes all variants of a specific *ProcessingUnit* or *Bus* out of the set of all *ProcessingUnit*s and *Bus*es ($VAR \subset ProcessingUnits$ or $VAR \subset Buses$) and the set T of all *Task*s of the Software Architecture.

$$(\sum\nolimits_{\{v \in VAR | \exists t \in T.allocate(t)=p\}} 1) \leq 1 \tag{6.7}$$

The proposed variation point semantic is that either one specific variant $v \in VAR$ is chosen (at least one *Task* or *Signal* is allocated to exactly one *ProcessingUnit* or *Bus* variant) or none (no *Task* or *Signal* is allocated to any variant).

Considering the introduced Basic Patterns (Topology and Variability Pattern), we are now able to define an E/E Architecture Exploration problem. The Constraint and Objective Patterns which we first introduced in [21] and which will be presented in the following, can now be additionally added in order to meet specific automotive requirements which have to hold for an E/E Architecture.

6.2.2. Constraint Patterns

In the following, we present automotive specific patterns depicting typical constraints in this domain, however, are also applicable in other domains. Those patterns were identified in the collaboration with Continental.

6.2.2.1. Allocation/Dislocation Pattern

The Allocation/Dislocation Pattern defines whether one specific *Task* $t \in T$ or one specific *Signal* $s \in S$ must to be allocated (cf. eq. 6.8) or must not be allocated (cf. eq. 6.9) to a *ProcessingUnit* $p \in P$ or *Bus* $b \in B$.

$$\begin{aligned} &t \in T, p \in P.allocate(t) = p \\ &s \in S, b \in B.allocate(s) = b \end{aligned} \tag{6.8}$$

$$t \in T, p \in P.\neg(allocate(t) = p)$$
$$s \in S, b \in B.\neg(allocate(s) = B) \tag{6.9}$$

The Allocation/Dislocation Pattern is one of the most frequently used patterns in the automotive domain. There are many software artifacts such as *Task*s which have to run on a certain *ProcessingUnit*, due to legacy system configurations or due to the fact that one department is developing the software (*Task*s) for a specific *ProcessingUnit*.

6.2.2.2. Function Coupling/De-Coupling Pattern

The Function Coupling Pattern ensures that a set of certain *Task*s must be allocated to the same *ProcessingUnit*. Therefore, the user is choosing a set of *Task*s $T' \subseteq T$, whereby each $t \in T'$ has to be allocated to one *ProcessingUnit* $p \in P$.

$$\exists p \in P, \forall t \in T'.allocate(t) = p \tag{6.10}$$

The Function De-Coupling Pattern ensures that a set of user defined *Task*s $T' \subseteq T$ must not be allocated to the same *ProcessingUnit*. This entails, that every *Task* defined in this pattern is allocated to a different *ProcessingUnit*.

$$\forall p_1 \in P, \forall p_2 \in P, \forall t_1 \in T', \forall t_2 \in T'.$$
$$(allocate(t_1) = p_1) \wedge (allocate(t_2) = p_2) \Rightarrow (p_1 \neq p_2) \vee (t_1 = t_2) \tag{6.11}$$

The Function De-Coupling Pattern is mainly used in order to describe safety requirements (a group of *Task*s which have to or must not run on the same *ProcessingUnit*).

6.2.2.3. Safety Pattern

In this pattern, the user is restricting the *Allocation* of *Task*s using their ASIL value. For each *Task* $t \in T$ it has to be determined, which *ProcessingUnit* $p \in P$ it may be allocated to. A *Task* may only be allocated to a *ProcessingUnit*, if its ASIL is not higher than the ASIL of the *ProcessingUnit*.

$$\forall t \in T, \forall p \in P.(allocate(t) = p) \Rightarrow (safetyLevel(t) \leq safetyLevel(p)) \tag{6.12}$$

Safety-compliant deployment is one way to ensure that safety critical *Task*s are allocated to *ProcessingUnit*s which are reliable enough. This constraint is derived from the ISO26262 [25] standard.

6.2.2.4. Memory Pattern

The Memory Pattern limits the amount of *Task*s $t \in T$ which can be allocated to a *ProcessingUnit* $p \in P$ based on the memory provided by this *ProcessingUnit*. Considering equation 6.13, the sum of the memory of all *Task*s allocated to a certain *ProcessingUnit* p may not exceed the memory provided by the this *ProcessingUnit*.

$$\forall p \in P.(\sum\nolimits_{\{t \in T | allocate(t)=p\}} memory(t)) \leq memory(p) \tag{6.13}$$

The Memory Pattern can be used to define constraints for read-only memory (flash) or random access memory (RAM) which would consequently result in two Constraint Patterns. The required flash memory of a *Task* is in general known during development time. The required RAM can be estimated or can be determined empirically.

6.2.3. Objective Patterns

In addition to *Constraint Pattern*s, which are used to restrict the set of possible solutions, it is necessary to define *Optimization Pattern*s. An *Optimization Pattern* is used to search for an optimized E/E Architectures. By nature, the number of objectives is not limited, which allows for multi-objective optimization. An Objective Pattern consists of an "optimization direction" (maximize or minimize) and an *AggrFunExpr* (optimizing the expression in the chosen optimization direction).

6.2.3.1. Property Objective Pattern

The goal of this pattern is to minimize or maximize a certain system level property of the Hardware Resources which form the Hardware Topology. A certain property of all *ProcessingUnit*s P is therefore summed up, if at least one *Task* $t \in T$ is allocated to *ProcessingUnit* $p \in P$ ($\{p \in P | \exists t \in T.allocate(t) = p\}$). Expression 6.14 describes the *Objective Pattern* for any property of a *ProcessingUnit* $p \in P$. For instance, a minimization means that if we can omit one *ProcessingUnit* with a certain $property(p)$ (by not allocating any *Task* to it) we can reduce the the overall sum of all properties of our E/E Architecture (e.g. cost, power, ...).

$$min/max(\sum\nolimits_{\{p \in P | \exists t \in T.allocate(t)=p\}} property(p)) \tag{6.14}$$

One example for such an objective is the minimization of costs of Hardware Resources. This means, that omitting one *ProcessingUnit* with a certain cost can reduce the cost of the whole E/E Architecture. All other available properties are described by the *ModelElemProperty* expression in the DSML (Table 6.1).

6.2.3.2. Cardinality Objective Pattern

The goal of this pattern is to minimize the amount (cardinality) of *ProcessingUnit*s which are used in the E/E Architecture independent from their properties. A *ProcessingUnit* is referred to as used, if at least one *Task* is allocated to it.

$$min\,|\{p \in P | \exists t \in T.allocate(t) = p\}| \tag{6.15}$$

6.2.3.3. Bandwidth Objective Pattern

The Bandwidth Objective Pattern is a modified version of the Property Objective Pattern. It is minimizing the busload of one specific *Bus* b_{min}. The busload is estimated by summing up the size of all *Signal*s (defined at the output port of the sending *Task*) which are allocated to this specific bus b_{min}. This sum is then minimized.

$$min(\sum\nolimits_{\{s \in S | allocate(s) = b_{min}\}} size(s)) \tag{6.16}$$

The goal of this pattern is to minimize the bandwidth of a specific *Bus* in order to provide an estimate whether it can be minimized to zero such that this *Bus* might be omitted.

7. Exploration Viewpoint

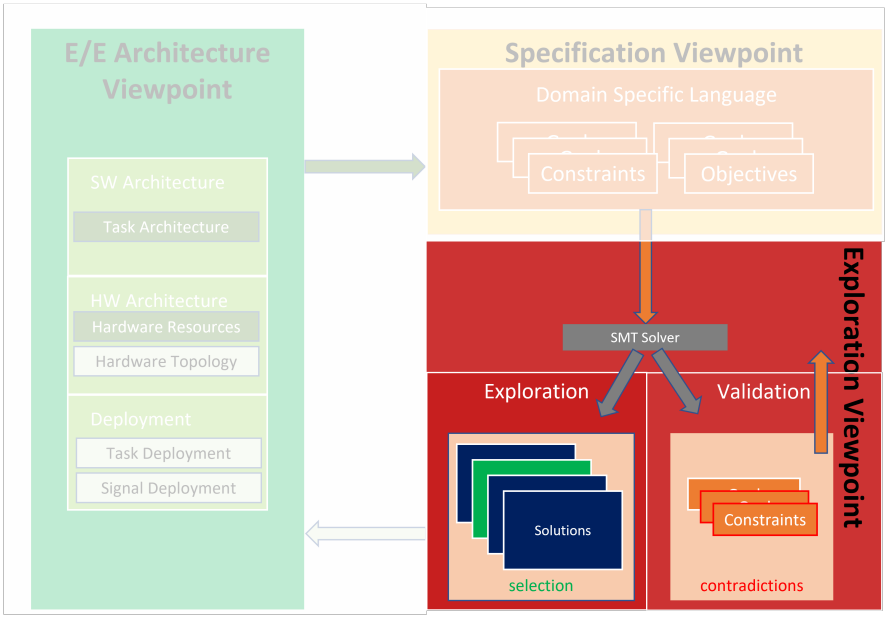


Figure 7.1.: Overview of viewpoints

The Exploration Viewpoint describes the formal E/E Architecture exploration problem definition using the language introduced in Section 6 and its translation into SMT (Satisfiability Modulo Theories) in order to be solved by a corresponding solver.

In Section 7.1 we define the E/E Architecture exploration problem. Section 7.2 will then introduce a meta-model to store the exploration problem definition. Section 7.3 shows the transformation of the E/E Architecture exploration problem expressed in the DSL into SMT. Lastly, in Section 7.4 we describe the results after a validation and exploration and how different solutions can be compared.

This Chapter has been published in Eder et al. [24] in a first version and has been taken over, adapted and considerably extended in this thesis.

7.1. E/E Architecture exploration problem definition

An E/E Architecture exploration problem E is defined as a tuple

$$E =< C, O >$$

where

1. C is the set of all constraints,
2. O is the set of all optimization objectives.

On the one hand, the set of constraints consists out of the basic patterns introduced in Section 6.2.1, the topology pattern and the variability pattern. On the other hand, it consists of the constraint patterns introduced in Section 6.2.2 which can be individually defined for any exploration.

We therefore define the set of Constraints C as

$$C :=\{c_{top}, c_{con_0}, ..., c_{con_i}, c_{v_0}, ..., c_{v_j}, c_{prop_0}, ..., c_{prop_k}, c_{pat_0}, ..., c_{pat_n}\}$$

where

1. c_{top} is the topology pattern constraint as defined in Section 6.2.1.1,
2. $c_{con_0}, ..., c_{con_i}$ are i connection constraint defining the *Connector*s of all *ProcessingUnit*s and *Bus*es via the *connectedP* and *connectedB* function (as defined in function definition 6.1) and 6.2 in Section 6.2.1.1).
3. $c_{v_0}, ..., c_{v_j}$ are j variability constraints defined in the variability pattern in Section 6.2.1.2. Each abstract variation point is formalized by exactly one variability constraint.
4. $c_{prop_0}, ..., c_{prop_k}$ are k property constraints determining the property value of each *ModelElement* by using the *property* function. (as defined by the $ModelElemPropertyFun$ in Section 6.1.6)
5. $c_{pat_0}, ..., c_{pat_n}$ are n individually - for each E/E Architecture exploration problem - defined constraints by using the constraints patterns introduced in Section 6.2.2.

The set of objectives consists out of the objective patterns introduced in 6.2.3 and can also be defined individually for each exploration.

We therefore define the set of Objectives O as

$$O := \{o_{pat_0}, ..., o_{pat_m}\}$$

where

$o_{pat_0}, ..., o_{pat_m}$ are m individually - for each E/E Architecture exploration problem - defined optimization objectives by using the objective patterns introduced in Section 6.2.3.

7.2. DSE Meta-Model

As introduced in the last Section, an E/E Architecture exploration problem consists out of constraints, which limit the set of possible solutions, and optimization objectives, which optimize this limited set of solutions towards a certain directions.

Figure 7.2 illustrates a meta-model capable of storing such an E/E Architecture exploration problem. Built upon the meta-model proposed in [87], an *ExplorationProblem* is described as a set of *ExplorationTarget*s. An *ExplorationTarget* is either an *ExplorationConstraint* or an *ExplorationObjective*. Each *ExplorationTarget* can be of a certain *Category*. In Eder et al. [21], we identified five different categories which could also be further extended.

- **Allocation** Predefined *Allocation*s of *Task*s/*Signal*s to *Hardware Resources* which may not be changed, due to, e.g., required hardware interfaces.
- **Memory** Describing constraints and objectives concerning the overall memory consumption of the *Hardware Resources*.
- **Safety** Describing Automotive Safety Integrity Level (ASIL) [25] constraints which have to be considered by a *Allocation*.
- **Cost** Describing constraints and objectives considering the costs of *Hardware Resources*.
- **Energy** Describing constraints and objectives regarding the energy consumption of the *Hardware Resources*.

Furthermore, an *ExplorationProblem* consists of an arbitrary number of *RuleSet*s. A *RuleSet* can by either of type *ConstraintRuleSet* or *ObjectiveRuleSet*. *RuleSet*s enable structuring an E/E Architecture exploration problem. They always contain a subset of references to *ExplorationTarget*s contained in the *ExplorationProblem*. A *RuleSet* may thus describe different variations of an exploration problem which may result in different solutions. This enables a system designer to compare the solutions of differing *RuleSets*, which all refer to the same system model.

It does furthermore not force a system designer to use all of the constraints and objectives. The concept of a *RuleSet* is especially helpful when contradicting constraints (respectively contradicting requirements) are detected. It enables a system designer to select and deselect constraints from a *RuleSet*. Such a relaxation leverages the handling of contradicting requirements as the constraints are derived from the requirements of the system.

Thus, the meta-model in Figure 7.2 shows how we can categorize and store an E/E Architecture exploration problem.

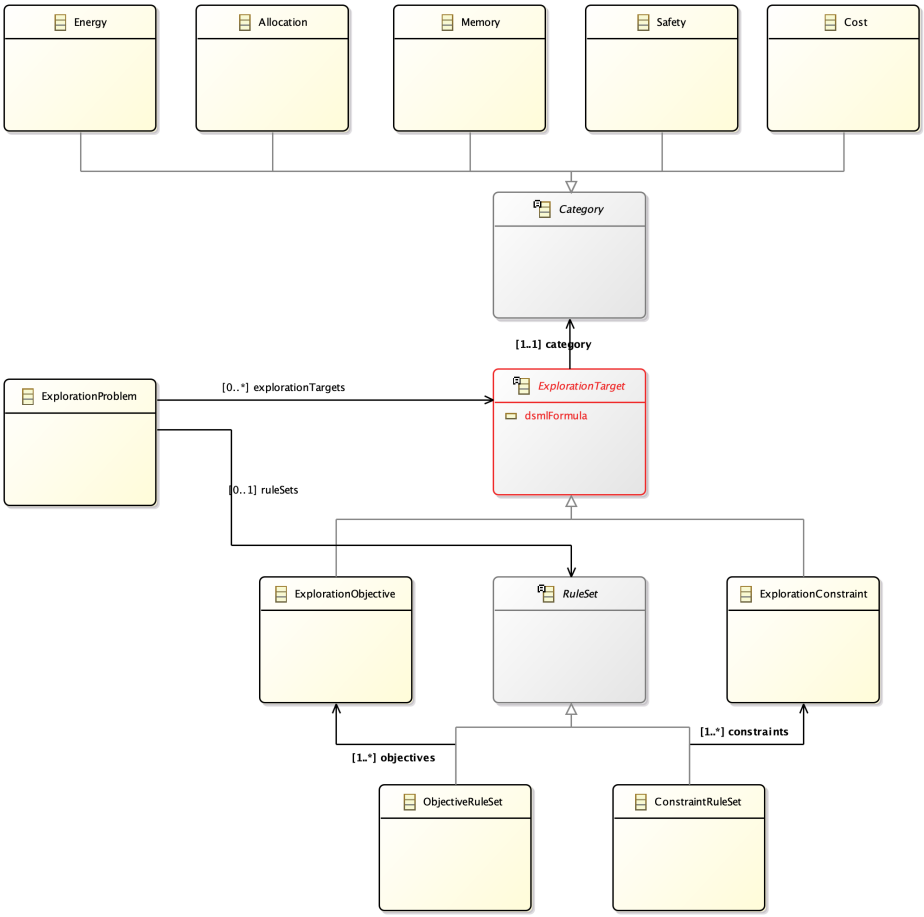


Figure 7.2.: Exploration Meta-Model

7.3. Translation into SMT

The model element sets T, S, P, B, C (*Task*s, *Signal*s, *ProcessingUnit*s, *Bus*es, *Connector*s; as defined in the beginning of Section 6.2) are translated into enumerations in SMT. Therefore, we use the SMT *declare-datatypes* expression for non-parametric datatypes (see 0 after Name declaration (*Task 0*)). Exemplarily, three enumeration types are defined in the following for every enumeration type.

```
(declare-datatypes ((Task 0)) (((task_1) (task_2) (task_3))))
(declare-datatypes ((Signal 0)) (((signal_1) (signal_2) (signal_3))))
```

```
(declare-datatypes ((ProcessingUnit 0)) (((pu_1) (pu_2) (pu_3))))
(declare-datatypes ((Bus 0)) (((bus_1) (bus_2) (bus_3))))
(declare-datatypes ((Connector 0)) (((con_1) (con_2) (con_3))))
```

All of the *ModelElementFunExpr* which are defined in the DSML are translated using an uninterpreted function of SMT.

```
(declare-fun start (Signall) Task)
(declare-fun end (Signal) Task)
(declare-fun allocate (Task) ProcessingUnit)
(declare-fun allocate (Signal) Bus)
(declare-fun connectedP (Connector) ProcessingUnit)
(declare-fun connectedB (Connector) Bus)
```

For every *ModelElementProperty* which is used in the exploration, an uninterpreted function is defined.

```
(declare-fun safetyLevel (Task) Int)
(declare-fun safetyLevel (ProcessingUnit) Int)
(declare-fun ram (Task) Int)
(declare-fun ram (ProcessingUnit) Int)
(declare-fun flash (Task) Int)
(declare-fun flash (ProcessingUnit) Int)
(declare-fun power (ProcessingUnit) int)
(declare-fun cost (ProcessingUnit) int)
(declare-fun size (Signal) int)
```

The topology pattern constraint c_{top} (Section 6.2.1.1) is translated into a quantified SMT formula. It refers to the datatypes defined above.

```
(assert
(forall ((s0 Signal))
(exists ((b4 Bus))
(forall ((e1 ProcessingUnit))
(forall ((e2 ProcessingUnit))
(exists ((c3 Connector))
(exists ((c5 Connector))
(let ((a!1 (and (= (allocate (f_start s0)) e1)
(= (allocate (f_end s0)) e2)))
(a!2 (xor (and (not (= e1 e2))
(= (connectedP c3) e1)
(= (connectedB c3) b4)
(= (connectedP c5) e2)
(= (connectedB c5) b4)
(= (allocate s0) b4))
(and (= e1 e2) (= (allocate s0) bus_null)))))
(=> a!1 a!2)))))))))
```

The connected constraints $c_{con_0}, ..., c_{con_i}$ are translated using the respective functions for each *ProcessingUnit* and *Bus*. The special *Connector* semantics introduced in Section 6.2.1.1 are taken care of by an *or* expression.

```
(assert (and (= (connectedP connector_1) pu_1)))
(assert (and (or (= (connectedB connector_1) bus_1)
(= (connectedB connector_1) bus_2))))
...
```

The variability pattern constraints $c_{v_0}, ..., c_{v_j}$ (Section 6.2.1.2) are translated for every *ProcessingUnit* or *Bus* which is a variation point (definition 6.7) into SMT using an if-then-else (*ite*) expression in SMT.

```
(assert (let ((a!1 (ite (or (= (allocate task_1) pu_1)
(= (allocate task_2) pu_1)
(= (allocate task_3) pu_1))
1
0))
(a!2 (ite (or (= (allocate task_1) pu_2)
(= (allocate task_2) pu_2)
(= (allocate task_3) pu_2))
1
0)))
(<= (+ a!1 a!2) 1.0)))
```

The properties of *ModelElements* $c_{prop_0}, ..., c_{prop_k}$ are translated using the respective functions for each *ModelElement*.

```
(assert  (= (safetyLevel task_1) 2)) ; ASIL B
(assert  (not (= (ram pu_1) 512))
...
```

The constraint patterns $c_{pat_0}, ..., c_{pat_n}$ (Section 6.2.2) are translated as follows.

The allocation pattern (Section 6.2.2.1) demanding the *Allocation* of a *Task task_1* onto a *ProcessingUnit pu_1* is translated as stated below. The dislocation would be translated accordingly with a negation.

```
(assert  (= (allocate task_1) pu_1))
(assert  (not (= (allocate task_1) pu1))
...
```

The function coupling pattern and de-coupling patterns (Section 6.2.2.2) are translated as follows.

The function coupling pattern (cf. definition 6.10) is translated by unfolding both quantifiers iterating over *Task*s and *ProcessingUnit*s.

```
(assert (or (and (= (allocate task_1) pu_1)
(= (allocate task_2) pu_1))
(and (= (allocate task_1) pu_2)
(= (allocate task_2) pu_2))
(and (= (allocate task_1) pu_3)
(= (allocate task_2) pu_3))))
```

The function de-coupling pattern (cf. definition 6.10) is also translated by unfolding the four quantifiers iterating over both *Task* sets and *ProcessingUnit* sets. Due to reasons of readability, we only show part of the SMT formulation here.

```
(assert (let ((a!1 (=> (and (= (allocate task_1) pu_1)
(= (allocate task_1) pu_1))
(or (not (= pu_1 pu_1))
(= task_1 task_1))))
(a!2 (=> (and (= (allocate task_1) pu_1)
(= (allocate task_2) pu_1))
(or (not (= pu_1 pu_1))
(= task_1 task_2))))
(a!3 (=> (and (= (allocate task_2) pu_1)
(= (allocate task_1) pu_1))
(or (not (= pu_1 pu_1))
(= task_2 task_1))))
...
(and a!1 a!2 a!3 ...)))
```

The safety pattern (Section 6.2.2.3) is also translated by unfolding both quantifiers iterating over *Task*s and *ProcessingUnit* sets. Due to reasons of readability, we also only show the SMT formulation for *ProcessingUnit pu_1* here.

```
(assert (and (=> (= (allocate task_1) pu_2)
(<= (safetyLevel task_1)
(safetyLevel pu_2)))
(=> (= (allocate task_2) pu_2)
(<= (safetyLevel task_2)
(safetyLevel pu_2)))
(=> (= (allocate task_3) pu_2)
(<= (safetyLevel task_3)
(safetyLevel pu_2)))))
(assert (and (=> (= (allocate task_1) pu_1)
...
```

The memory pattern (Section 6.2.2.4) is also translated by unfolding both quantifiers iterating over *Task*s and *ProcessingUnit* sets. The sum is calculated by using the if-then-else statement in SMT which returns zero if the respective *Task* is not allocated to the respective *ProcessingUnit* (otherwise the value returned by the memory function). Due to reasons of readability, we also only show the SMT formulation for *ProcessingUnit pu_1* here. We are illustrating the translation for RAM memory here. The translation for flash memory is accordingly.

```
(assert (let ((a!1 (+ (ite (= (allocate task_1) pu_1)
(ram task_1)
0)
(ite (= (allocate task_2) pu_1)
(ram task_2)
0)
(ite (= (allocate task_3) pu_1)
(ram task_3)
0))))
(and (<= a!1 (ram pu_1)))))
(assert (let ((a!1 (+ (ite (= (allocate task_1) pu_2)
...
```

The objective patterns $o_{pat_0}, ..., o_{pat_m}$ (Section 6.2.3) are translated as follows.

The property objective pattern (Section 6.2.3.3) is translated into SMT by summing up all property values of *ProcessingUnit*s where at least one *Task* is allocated to. This sum is then minimized.

```
(minimize (let ((a!1 (ite (or (= (allocate task_1) pu_1)
(= (allocate task_2) pu_1)
(= (allocate task_3) pu_1))
(power pu_1)
0))
(a!2 (ite (or (= (allocate task_1) pu_2)
(= (allocate task_2) pu_2)
(= (allocate task_3) pu_2))
(power pu_2)
0))
(a!3 ...
...
(+ a!1 a!2 a!3 ... )))
```

The translation of the bandwidth objective pattern (Section 6.2.3.3) needs a help function and constraint. The help function *weight* is used to calculate the weight of the *Bus* which

shall be minimized. The constraint defines the value of the weight function by summing up all the sizes of all *Signal*s which are allocated to the *Bus* to be minimized. The weight function is then used to create the objective.

```
(declare-fun weight (BusImpl) Real)

(assert (let ((a!1 (+ (ite (= (allocate signal_1) bus_1)
(size signal_1)
0)
(ite (= (allocate signal_2) bus_1)
(size signal_2)
0))))
(and (= (weight bus_1) a!1))))

(minimize (weight bus_1))
```

The cardinality objective pattern (Section 6.2.3.2) is translated similarly to the property objective pattern, except that it only counts the *ProcessingUnit*s where at least one *Task* is allocated to. This value is then again minimized.

```
(minimize (let ((a!1 (ite (or (= (allocate task_1) pu_1)
(= (allocate task_2) pu_1)
(= (allocate task_3) pu_1))
1
0))
(a!2 (ite (or (= (allocate task_1) pu_2)
(= (allocate task_2) pu_2)
(= (allocate task_3) pu_2))
1
0))
(a!3 ...
...
(+ a!1 a!2 a!3 ...)))
```

7.4. E/E Architecture exploration solutions

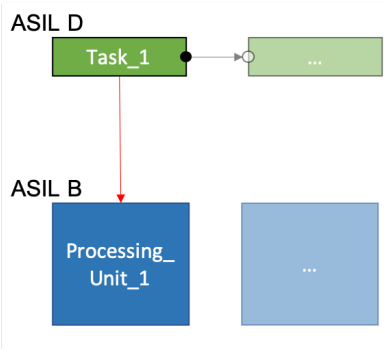
Considering the translated constraints and objectives into SMT, they can now be solved by an SMT solver. We are using the Z3 SMT solver [1]. When solving such a problem, we distinguish between two different types as depicted in the overview in Figure 7.1, Validation and Exploration. Validation focuses on finding contradicting constraints which may give a hint at contradicting requirements and will be described in more detail in Section 7.4.1. Exploration focuses on finding different optimized E/E Architectures entailing the exploration of *Hardware Topologies* together with a *Task* and *Signal* Deployment and will be described in more detail in Section 7.4.2.

7.4.1. Validation

During validation we focus on finding contradicting constraints. Those contradictions may give a hint about contradicting requirements. Regarding Figure 7.3 for instance, there is

[1] https://github.com/Z3Prover

a predefined *Allocation* of a certain *Task* to a certain *ProcessingUnit*, e.g., due to the fact that the *ProcessingUnit* and the functionality of the *Task* are developed by the same department. This would be enforced by defining an allocation constraint (cf. Section 6.2.2.1). Furthermore, a safety constraint would be defined because the resulting architecture has to apply to ISO 26262 (cf. Section 6.2.2.3). This means that all *Task*s ASIL has to be smaller or equal to the ASIL of the *ProcessingUnit*. If the allocation constraint contradicts this safety constraint (e.g. if the ASIL of the *Task* is higher than the ASIL of the *ProcessingUnit* it has to be allocated to) there would not be any solution. Although this example may sound trivial, we have to imagine that industrial sized automotive architectures have sizes of at least 100 *Task*s and 60 *ProcessingUnit*. Finding such contradictions is a manually almost unsolvable *Task*, especially, considering that there are many more constraints which also interfere with each other.



1. Allocation Constraint (red arrow):
   $allocate(Task_1) = ProcessingUnit_1$
2. Safety Constraint:
   $allocate(Task_1) = ProcessingUnit_1 \Rightarrow$
   $(safetyLevel(Task_1) \leq$
   $safetyLevel(ProcessingUnit_1))$

Remark: we left out quantifiers in the formulas above in order to increase the readability.

Figure 7.3.: The scheme on the left side is showing one *Task* called $Task_1$ and one *ProcessingUnit* called $ProcessingUnit_1$. On the right side we are considering to exemplary constraints which have to hold: (1) an allocation constraint (red arrow on the left) and (2) a safety constraint.

In order to find such contradictions, we are using the unsat-core feature of the Z3 SMT-Solver[1]. This features returns a minimal set of contradicting constraints which has to be resolved in order to get a solution. Coming back to the example mentioned above, there are two contradicting constraints, an allocation constraint and a safety constraint. There may exist a variety of reasons for this contradictions such as

1. The contradiction is caused by contradicting requirements, as the constraints are derived from the requirements of the system.

   In the example this means, that we defined a safety constraint according to a safety requirement which demands compliance to ISO 26262. Moreover, the allocation constraint was defined because of a requirement demanding that the functionality of *Task_1* has to be executed on *Processing_Unit_1*. This entails that those requirements

are contradicting each other.

2. The contradiction is caused by changed properties in the Software Architecture or the Hardware Architecture.

   Considering that parts of the Software and Hardware Architecture and also constraints are reused in order to build a new system, there might be small changes. In the example, this could be caused by a changed ASIL of *ProcessingUnit* or *Task* which causes a contradiction. Regarding that there was no contradiction before, the ASIL of Processing_Unit_1 may have been lowered from ASIL D to ASIL B or the ASIL of Task_1 may have been increased frmo ASIL B to ASIL D.

3. Contradictions are hiding each other.

   This means that through resolving a contradiction a new contradiction appears. In the example, this could happen, for example, if we change the ASIL of $ProcessingUnit_1$ to ASIL D (e.g. due to the fact that we changed to a different vendor). Considering that we have an additional memory constraint demanding that the memory usage of *Task*s may not exceed the provided memory by the *ProcessingUnit*, this memory constraint may now cause a new contradiction which was hidden before. This means that that the contradiction between allocation and safety constraint was hiding the contradiction to the memory constraint. When trying to resolve this contradiction we have to keep in my mind that through changing the ASIL of $ProcessingUnit_1$ to D in the first place, changed the whole problem, as any *Task* can now be allocated to this unit because of ASIL D being the highest possible criticality level. It requires a good knowledge about the system in order to backtrack and resolve those contradictions.

Thus, resolving contradictions is non-trivial and has to be manually performed by the Exploration Engineer and the System Architect.

7.4.2. Exploration

During exploration we focus on finding an optimized or even optimal E/E Architecture by means of a *Hardware Topology* together with a *Task* and *Signal* Deployment. This step is performed after a successful validation which entails that there are no contradicting constraints. In particular, the optimization objectives are in focus here. In order to calculate optimized solutions, we are using the optimizing solver of the z3 SMT Solver [49] (with the pareto option). The goal of the exploration is to provide different optimized solutions to the System Architect, among which he can then choose the best one. As the optimization objectives are not independent the solutions are pareto optimal and the architect has to manually resolve this trade-off.

In the following, we will first describe metrics which characterize an E/E Architecture. Secondly, we will illustrate how different solutions are compared (based on the introduced metrics) supporting the System Architect in the decision process.

7.4.2.1. Metrics

In order to compare different architecture solutions, we want to describe metrics which characterize them. The following list is non-exhaustive and is based on the optimization objectives which can optimize those metrics (see also the Property Objective Pattern in Section 6.2.3.1). There maybe more metrics added in order to describe a solution in more detail.

1. Cost

   Describes the overall sum of costs of all *ProcessingUnit*s in the *Hardware Topology*.

2. Number of *ProcessingUnit*s/buses

   Describes the number of *ProcessingUnit*s/*Bus*es in the *Hardware Topology*.

3. Weight

   Describes the weight of the *Hardware Topology* as the overall sum of weights of all *ProcessingUnit*s.

4. Power consumption

   Describes the power consumption of the *Hardware Topology* as the overall sum of power consumption of all *ProcessingUnit*s.

5. Memory usage

   Describes the memory usage in the *Hardware Topology* as the overall sum of memory (either flash or ram memory) of all *ProcessingUnit*s.

6. Busload

   Describes the busload of a certain *Bus* in the *Hardware Topology* as the sum of *Signal* sizes which are sent via this *Bus*.

7.4.2.2. Solution comparison

Considering the metrics which quantify the different *Hardware Topology* solutions and thus an E/E Architecture, we can now compare them. On the one hand, we are using a spider chart representation which qualitatively shows the difference between the different *Hardware Topologies* and, on the other hand, a simple table which quantitatively shows the difference. In order to illustrate how different *Hardware Topologies* can be compared, we have a closer look at three exemplary solutions. Those solutions were generated out of a set of *Hardware Resources* which are schematically depicted in Figure 7.4. There are three *ProcessingUnit*s and two *Bus*es which each have two different variants. During the exploration one of the variants is chosen or none and an optimal connection between *ProcessingUnit*s and *Bus*es is calculated.

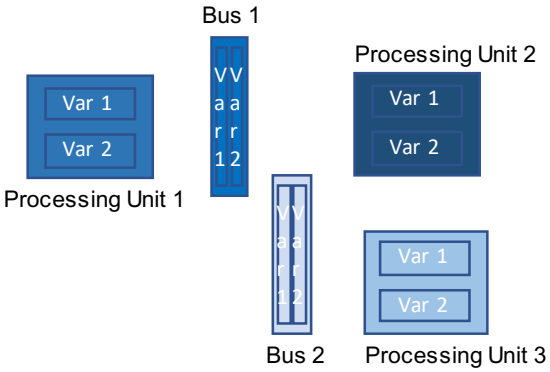


Figure 7.4.: Exemplary set of *Hardware Resources* consisting of 3 *ProcessingUnit*s and 2 *Bus*es each having 2 variants.

Figure 7.5 shows three possible solutions out of the *Hardware Resources* shown in Figure 7.4. The spider chart representation on top shows the qualitative comparison between three solutions according to the metrics introduced in 7.4.2.1. On the bottom the three different *Hardware Topologies* are depicted, where the red colored variants show the variants which have been chosen during the exploration.

The trade-off between the solutions can easily be seen as Topology 2 is dominating each other solution except in the busload direction. This makes sense as Topology 2 only uses one *Bus* which thus has to realize the whole communication of *Signal*s whereas both other solutions use two *Bus*es which results in a more distributed communication.

Considering Topology 1 and 3 we can also see a trade-off. Where Topology 1 is more optimal considering cost and weight, Topology 3 is more optimal considering power consumption, Flash Memory and Busload.

	E/E Arch 1	E/E Arch 2	E/E Arch 3
Cost	9	8	11
Weight	11	9	13
Power Consumption	12	7	11
Flash Memory	14	9	12
Busload	12	20	11
Number of processing units	9	6	9

Table 7.1.: Table visualization of metrics

Table 7.1 shows the same solutions as illustrated in Figure 7.5 in a table representation. Here, the trade-off between the solutions can not be seen as easily as in the spider chart. However, the table gives the exact number of each metric. By that, the System Designer and the Exploration Engineer can decide which solution will be chosen.

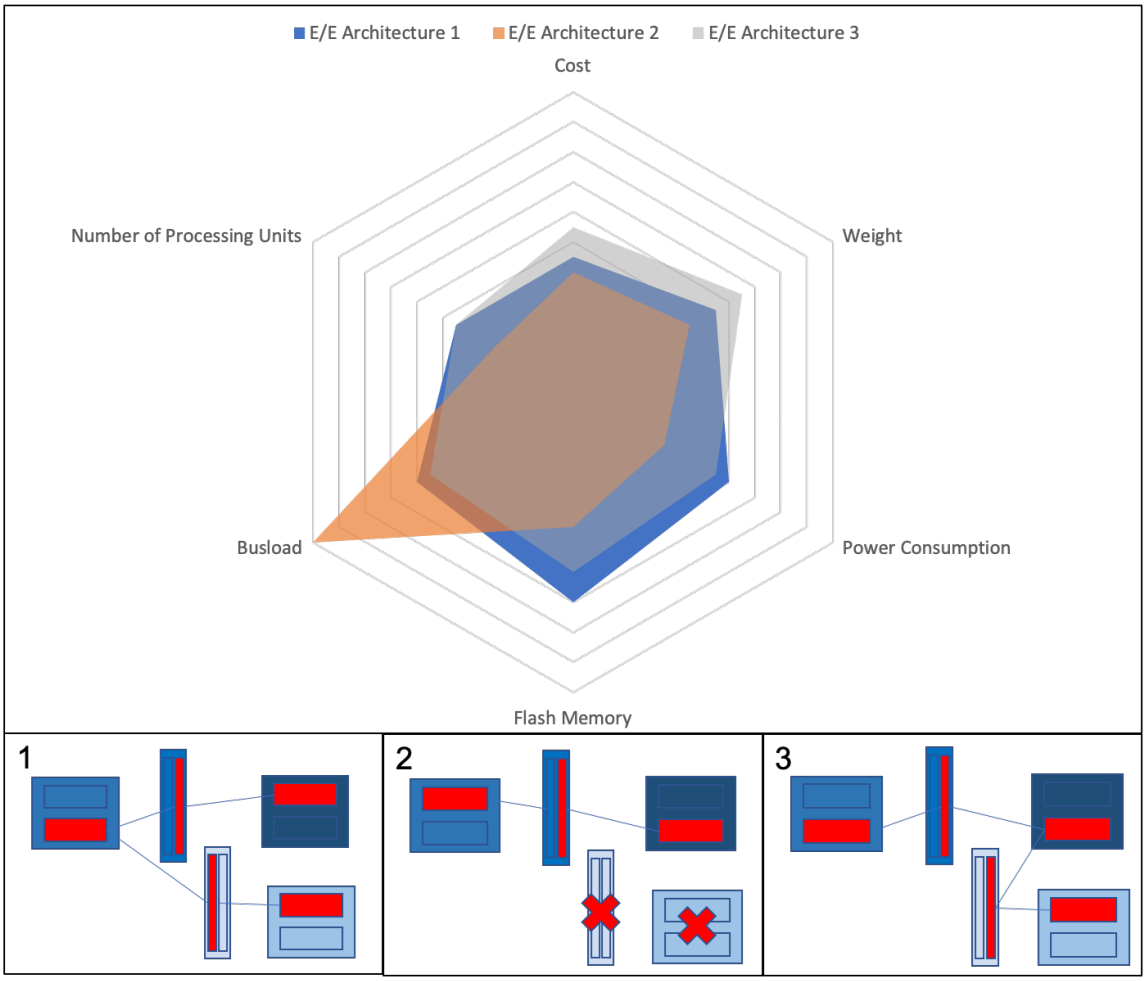


Figure 7.5.: Spider-chart visualization of metrics of three different *Hardware Topologies*

8. Evaluation

Throughout an ongoing collaboration with an automotive Tier-1 supplier, resulting in three publications [21] (MODELS 17), [23] (MODELS 18) and [24] (SoSyM Journal) we gained insights into the state of practice, considering the configuration of several concrete E/E Architectures. In the following, we evaluate this work against the state of practice. In particular, we compare the combination of the presented viewpoints and the proposed methodology against the industrial practice (Chapters 4-7).

We evaluate this thesis by setting up key performance indicators (KPIs), namely a set of evaluation criteria, enabling a comparison of the state of practice against the proposed E/E Architecture exploration methodology (cf. Section 8.1). Afterwards, we present two case studies conducted in an industrial collaboration focusing on solving the Deployment onto E/E Architectures as well as on exploring E/E Architectures (cf. Section 8.2).

8.1. Evaluation criteria and comparison to state of practice

In the following, we describe criteria which serve as means to evaluate this thesis. The following 6 KPIs are defined to compare the creation of E/E Architectures in industrial practice against the proposed approach of this thesis.

K1: Execution type [manual, semi-automatic, automatic]

K2: Execution duration [short[h], medium[d], long[w]]

K3: Process type [static, iterative]

K4: Reaction speed [slow, fast]

K5: Optimization potential [little, considerable]

K6: Verification type [informal, formal]

Table 8.1 illustrates the results of this comparison which leads to individual conclusions. In each of the following Sections, we will at first describe the definition of the respective KPI. Based on this definition, we will rate the state of practice and the proposed exploration approach of this thesis. Lastly, we will explain how the proposed exploration approach improves the state of practice.

KPIs	State of Practice	Exploration
K1 - Execution type	manual	semi-automatic
K2 - Execution duration	long	short to medium
K3 - Process type	static	dynamic/iterative
K4 - Reaction speed	slow	fast
K5 - Optimization potential	little	considerable
K6 - Verification type	informal	formal

Table 8.1.: Result of the evaluation comparing the generation of E/E Architecture in practice against the proposed exploration approach.

8.1.1. K1 - Execution type

Description The KPI *execution type* refers to how steps in the development process can be performed. These steps (e.g. according to the V-Model) can be executed in the following three ways:

1. Manual:

   The *execution type* of a development step is completely manual. This entails that least one system architect performing all necessary development steps her-/himself. Considering the creation of an E/E Architecture, this means that all the models (cf. Chapter 5) are created manually, mostly based on implicit knowledge. For instance, either based on legacy designs or on personal experiences.

2. Semi-automatic:

   The *execution type* of a development step can be performed using a considerate amount of automation. This entails, that the development steps are partly performed automatically, supporting a responsible person. However, a reasonable amount of steps need to be performed in a manual way.

3. Automatic:

   The execution of a development step can be performed completely automated. All necessary development steps can be performed automatically involving no or almost no manual work. By nature, this requires a substantial confidence in the underlying mechanisms and tools.

KPI	State of Practice	Exploration
K1	manual	semi-automatic

State of practice Considering the state of practice, the creation of an E/E Architecture is performed **manual**ly, meaning that all of the models described in the E/E Architecture Viewpoint (cf. Chapter 5) are created manually. In particular, the Software Architecture (e.g. Task Architecture), the Hardware Architecture (especially the Hardware Topology) and its connection (using a Deployment) are created manually.

For example, regarding a Task Architecture consisting of only 30 *Task*s which have to be allocated to only 10 *ProcessingUnit*s already opens up a design space of 10^{30} *Allocation* possibilities (without consideration of any constraints). Additionally, the requirements which have to hold for such a Deployment have to be checked manually. This entails, for instance, checking if the memory of any *ProcessingUnit* suffices w.r.t. the *Task*s allocated to this *ProcessingUnit* or checking if any *Signal* can be allocated to a certain *Bus* w.r.t. to the bandwidth of that *Bus*.

Exploration This thesis proposes a **semi-automatic** exploration methodology, meaning that parts of the models described in the E/E Architecture Viewpoint (cf. Chapter 5) can be created/calculated automatically. In particular, this means that not only the *Hardware Topology* can be calculated automatically but also (and at the same time) the Deployment of *Task*s to *ProcessingUnit*s and *Signal*s to *Bus*es. Hereby, requirements (in the form of constraints) which have to hold, are automatically satisfied. The proposed exploration is semi-automatic, due to the fact that not all of the models described in the E/E Architecture Viewpoint are calculated automatically. Still, the solutions resulting from the automatic exploration (*Hardware Topology* and *Task*/*Signal* Deployment) have to be reviewed by a system architect.

Improvement to state of practice The semi-automatic exploration approach considerably improves the state of practice as it does not only automatize a process which has been done manually. It also provides means to cope with the future challenge of building E/E Architectures. These architectures include increasingly complex features like driver assistance and automated driving *Task*s while the *Hardware Topology* gradually gets more and more centralized. Regarding this rising complexity, the manual creation of these architecture gets a manually unsolvable task.

8.1.2. K2 - Execution duration

Description The KPI *execution duration* provides a measure of how long it takes to perform a certain development step. There are three types of time intervals, which we distinguish here.

1. Short [h]:

   We specify the *execution duration* of a certain development step as short if it takes 0 to 8 hours. As 8 hours is the usual unit to calculate a work day, we therefore specify

that the successful execution of a development step as short, if a system architect can execute this *Task* within one day.

2. Medium [d]:

   We specify the *execution duration* of a certain development step as medium if it takes 1 to 5 days. As 5 days is the usual unit to calculate a work week, we therefore specify the successful execution of a development step as medium if a system architect can execute this *Task* within one week.

3. Long [w]:

   We specify the *execution duration* of a certain development step as long if it takes more than 1 week and up to 4 weeks. As 4 weeks is the usual unit to calculate a man-month, we specify the successful execution of a development step as long, if a system architect cannot execute this development step within one week but within 4 weeks. Without loss of generality, we consider every development step to be dividable into parts which can be executed within 4 weeks at most.

KPI	State of Practice	Exploration
K2	long	short to medium

State of practice The execution duration in industrial practice is **long**. Taking into account that the execution type (cf. K1 in Section 8.1.1) is manual, not only the *Hardware Topology* has to be created manually but also the Deployment. As a small industrial sized architecture already consists of 31 *Task*s and 18 *ProcessingUnit*s [21] the manual creation of an E/E Architecture takes weeks even for an experienced architect. This is due to the fact that, he does not only have to check architectural constraints ("Is it possible to send a *Signal* between *Task* a and b if they are allocated onto *ProcessingUnit*s x and y?)" but also standard compliance such as compliance to ISO 26262 or resource requirements ("Is the memory of each processing sufficient?). Due to the intricate dependencies of both *Task* and *Hardware Topology*, a mistake in this manual process may cause a huge backtracking process even extending the duration of the process.

Exploration Considering the proposed exploration methodology, the execution duration is **short** to **medium**. Depending on the size of the architecture and if a validation or an exploration of E/E Architectures is considered, it may take a few minutes to a few hours. Especially, finding contradictions during validation phase typically takes only a few seconds to a few minutes even for big architectures. An exploration of E/E Architectures typically takes a few hours. For example, for the use case presented in [21] the calculation of a Pareto optimized solution took 60s. In the use case presented in [23] it took 2h to calculate Pareto optimized E/E Architectures.

Improvement to state of practice Due to the fact that, the execution duration is much shorter using the exploration approach (hours versus weeks), efficiency can be substantially increased. On the one hand, this may not be too surprising, as we are comparing a manual against a semi-automatic approach. However, on the other hand, the exploration can significantly reduce the manual workload of responsible system architects or teams as it can save at least a few days of work. Considering that, for example, such an exploration has to be performed quarterly, the exploration can even save weeks of work per year.

8.1.3. K3 - Process type

Description The KPI *process type* specifies, in which way the development steps are performed during the development of a system. We distinguish two basic types of processes here:

1. Static ("waterfall"):

   We specify the *process type* to be static, if the development steps have to be executed in a sequentially fixed order, with no or little possibility to be changed. This entails, that in case of a problem, the process has to be backtracked sequentially until the problem is solved. After its resumption, the same process steps which have just been backtracked, have to be performed again. The development process is thus statically defined which requires again well-defined additional processes to react to changes or problems. Hence, we refer to a waterfall-like process type here.

2. Iterative/Dynamic ("agile")

   We specify the *process type* to be iterative/dynamic if the development steps do not have to be executed in a sequentially fixed order. The development can easily react to changes and does not require static processes to act to changes or problems. Hence, we refer to an agile process type here.

KPI	State of Practice	Exploration
K3	static	agile/iterative

State of practice The type of process is **static**. Taking into account the insight we gained into industrial practice, there are two sequential steps which are performed: (1) creating and analyzing different possible *Hardware Topologies* and choosing the most suitable one, (2) creating the *Task* and *Signal* Deployment onto the chosen hardware topology. This static process is also a consequence of the manual execution type (K1 in Section 8.1.1) as, due to the complexity of today's systems, the two development steps cannot be performed manually at the same time. Moreover, this entails, that changes in either *Hardware Topology* or Deployment would require a further sequential execution of both process steps. This is also reflected in a slow reaction to changes (K4 in Section 8.1.4).

Exploration The proposed exploration methodology supports an **agile/iterative** process. On the one hand, due to the fact that, the two development steps of (1) creating and analyzing *Hardware Topologies* and (2) creating the *Task* and *Signal* Deployment onto the *Hardware Topologies* can be calculated at the same time. On the other hand, the approach proposes a methodology which enables an agile development which only requires another exploration iteration if a change occurs.

Improvement to state of practice The proposed agile exploration methodology considerably improves the state of practice. It is able to replace a static and sequential process of finding valid and also optimized *Hardware Topologies* and respective Deployments by an agile process which can perform both process steps at the same time. Due to the lightweight nature of the process, this enables many exploration iterations which eases the task of finding an E/E Architecture.

8.1.4. K4 - Reaction speed

Description The KPI *reaction speed* specifies, how fast the development process is able to react to sudden changes during the development. We distinguish two (extreme) types here:

1. Slow:

   We specify the *reaction speed* to be slow, if a change process has to be triggered in order to adapt to changes. Typically, this sets off a big chain of change requests which potentially involves different organizational departments. In the end, each single change has to be integrated into the E/E Architecture.

2. Fast

   We specify the *reaction speed* to be fast, if a change does not trigger a heavyweight change process. This means that a change can be covered by a light-weight process, involving only a few or even one department. Hence, the integration of the change into the E/E Architecture requires only a small effort.

KPI	State of Practice	Exploration
K4	slow	fast

State of practice The reaction speed to architectural changes in the E/E Architecture is **slow**, due to the fact that this triggers a (heavyweight) change process. As the E/E Architecture of a vehicle usually reflects the organization structure of a company (e.g. one department is responsible for one *ProcessingUnit* including its software (*Task*s)), a change in the architecture has to be handled by different departments, thus involving several people in the change process. E.g. if a *Signal Allocation* has to be changed from *Bus x* to *Bus*

y, this possibly affects all *ProcessingUnit*s (and thus departments) which are connected to *Bus y* because a new *Signal* is sent via this bus, but also *Bus x* because a *Signal* has been removed.

Exploration The proposed exploration approach enables a **fast** reaction to changes. Considering the example above, an Exploration Engineer would have to create a new *Signal Allocation* constraint and then run a new exploration iteration. If the change leads to a problem, the Exploration Engineer immediately gets the feedback in the form of contradicting constraints which point to the cause of the problem. Thus, the problem can be solved immediately.

Improvement to state of practice Due to the proposed new role of an Exploration Engineer, the reaction to a change can be improved significantly. On the one hand, because the responsibility for reacting to the change is not split over different people but can be done by one person who can then distribute the results of the change. This would improve a heavy change process to a lightweight process where a change can be reflected by a new exploration iteration. On the other hand, due to the fast reaction time to changes, the exploration can detect contradictions earlier and embraces change requests, which are otherwise avoided as far as possible, due to the fact that they trigger a heavyweight change process.

8.1.5. K5 - Optimization potential

Description The KPI *optimization potential* specifies, the possibility of optimizing the E/E Architecture during development. We distinguish two (extreme) types:

1. Little (optimization potential)

   We specify the *optimization potential* to be little, if the development process is not explicitly supporting an optimization of the E/E Architecture during development. In particular, this may be the case, if the development process is static (see K3 in Section 8.1.3) and manual (see K1 in Section 8.1.1). Due to the nature of the consequent process, an architectural optimization would only be possible by applying the optimization to the E/E Architecture and triggering a change process to verify the correctness of the optimized architecture. Considering a manual development process, this would require an immense effort of rework. In the worst case, this results in the reversion of the optimization, due to the fact that, the requirements of the system can no longer be satisfied.

2. Considerable (optimization potential)

   We specify the *optimization potential* to be considerable, if the development process is actively supporting an optimization of the E/E Architecture during development. In particular, this may be the case, in an agile (see K3 in Section 8.1.3) and at least

semi-automatic (see K1 in Section 8.1.1) process. This means, that applying an architectural optimization to the system does not trigger a static (possibly heavyweight) change process. An optimization is automatically verified within one development iteration. Even if the optimization is rejected, e.g., caused by contradictions with the requirements of the system, the effort was low, due to the nature of the agile and semi-automatic process.

KPI	State of Practice	Exploration
K5	little	considerable

State of practice The optimization potential for E/E Architectures is **little**. For example, if one would optimize the costs of an E/E Architecture, one way to optimize would be to reduce the *Hardware Topology* by one processing unit. This means, that the costs of this *ProcessingUnit* could be saved. However, this also means that the Deployment of *Task*s and *Signal*s onto the *Hardware Topology* has to be redone manually (K1 in Section 8.1.1), checking if the constraints of the system can still be satisfied. In the worst case, this is not possible and the optimization cannot be performed. As the process type is manual (K3 in Section 8.1.3), this takes a considerable amount of time which would have been in vain in the depicted case. Considering the insight we gained into industrial practice, the optimization is limited to choosing up front between different manually created *Hardware Topologies*. After one *Hardware Topology* has been chosen, the *Task/Signal* Deployment is done manually. As the Deployment is not considered during *Hardware Topology* creation, possible optimization potential is lost.

Exploration The proposed exploration methodology enables the formulation of optimization objectives that can be considered in every exploration iteration, thus, providing considerable optimization potential. Due to the fact that, *Task/Signal* Deployment and *Hardware Topology* are calculated at the same time, the resulting E/E Architecture is optimized considering the *Hardware Topology* and the Deployment.

Improvement to state of practice The improvement to the state of practice are twofold: On the one hand, the effort of optimization is reduced to the formulation of formalized optimization objective which are then automatically considered in every exploration iteration. On the other hand, the optimization is even more precise as it does consider the *Hardware Resources* and the *Task/Signal* Deployment at the same time.

For instance, it might be favorable to choose a more expensive *Hardware Resource* over a cheaper one. Assuming, an expensive sensor *ProcessingUnit*, this could mean that the sensor data can already be pre-processed on the unit itself. This pre-processed data is then sent via a *Bus* to a receiving task. In case of a cheaper sensor, that is not able to pre-process data, this leads to sending raw data via the bus. Consequently, this requires a significant

higher amount of bandwidth of the *Bus* which is connected to this sensor. Our exploration approach is automatically considering such decisions when optimizing E/E Architectures.

8.1.6. K6 - Verification type

Description The KPI *verification type* specifies, how the correctness of the E/E Architecture can be verified. We distinguish two types here:

1. Informal

   We specify the *verification type* as informal, if the requirements of the architecture can only be captured informally. This means, that the verification of the system under development has to be performed, e.g., by a manual review. This may involve multiple persons, resulting in an error-prone and opinion-based process. Furthermore, this consequently shifts efforts to later stages of development - e.g. testing - which could mean that only during testing, some of the problems become visible which have been overlooked in a review.

2. Formal

   We specify the *verification type* as formal if the requirements of the system can be captured formally. This means that the verification of the system under development can be done automatically. Hence, the process of verification is objective and discovers, e.g., contradictions of requirements. As such, the process requires less persons and enables verification already at early stages of development. This entails that problems can also be solved earlier. However, it still relies on correctly formalized requirements.

KPI	State of Practice	Exploration
K6	informal	formal

State of practice Requirements for E/E Architectures can only be captured **informal**ly and thus only be checked manually. This means that, for example, safety requirements, considering the adherence to ASILs, have to be enforced manually which may lead to erroneous architectures ("Is the ASIL of the *Task* allocated onto a *ProcessingUnit* small or equal to the processing unit's ASIL?"). Furthermore, resource requirements like the sufficiency of flash memory or RAM have to be checked ("is there sufficient memory on a *ProcessingUnit* to execute the allocated *Task*s?"). Errors in those manual checks which have to be done via reviews, may, in the worst case, only be discovered during testing of the system.

Exploration The proposed exploration methodology enables **formal** capturing of requirements for E/E Architectures. Thus, they can be checked automatically. For instance, safety

requirements or memory requirements, can be captured formally by an Exploration Engineer using constraint patterns as proposed (cf. Chapter 6). This does not only enable automatic checks of requirements but also early detection of contradictions.

Improvement to state of practice The possibility to formally capture requirements and also check them automatically is considerably improving the state of practice. This enables an early automatic verification of the E/E Architecture, which entails a detection of possible defects already at early stages of development. That also means that they are easier and less costly to solve. In the course of industrial collaborations, for instance, we were thus able to find contractions in existing E/E Architectures showing, e.g., contradictions in *Signal* Deployments which were pointing to inapplicable communication routes.

8.2. Exploration in industrial context

In the following, we describe two case studies which we conducted together with an automotive Tier-1 supplier. In Section 8.2.1, we will briefly describe a Deployment case study, where we explored different *Task*/*Signal* Deployments. This work was published in Eder et al. [21] on MODELS conference in 2017.

In Section 8.2.2, we will describe a E/E Architecture case study where different possible E/E Architectures (*Hardware Topologies* and *Task*/*Signa*l Deployment) are explored. This case study is based on two publications on MODELS conference 2018 [23] and in SoSyM Journal [24].

8.2.1. Deployment exploration study

We conducted a deployment exploration study with an automotive tier-1 supplier to show the applicability of exploration approaches in general. This work was mainly focused on exploring optimized *Task* and *Signal* Deployments. In this work, we could already show that an exploration approach can significantly improve the state of practice. Not only, because an exploration approach enables automation but also due to the fact that it provides an objective approach. This was in particular helpful for the tier-1 supplier as they could get rid of opinion-based solutions. For more detailed insights into this case study, we would like to refer to Eder et al. 2017 [21].

8.2.2. E/E Architecture exploration study

In this Section, we describe an E/E Architecture exploration study. We calculate a *Hardware Topology* based on a set of available *Hardware Resources* and the Deployment of *Task*s and *Signal*s to these *Hardware Resource*s. The study is an adapted form of the use cases presented in [23] and [24].

In Section 8.2.2.1, we describe the models according to the introduced E/E Architecture Viewpoint (Chapter 5). In Section 8.2.2.2, we describe the constraints and optimization

objectives according to the introduced Specification Viewpoint (Chapter 6). In Section 8.2.2.3, we describe the validation and exploration according to the introduced Exploration Viewpoint (Chapter 7) and discuss the E/E Architecture results.

8.2.2.1. Model

In the following, we describe the models which are the basis for calculating optimal E/E Architectures. On the one hand, a Task Architecture and, on the other hand, a Hardware Architecture in terms of *Hardware Resource*s.

Task Architecture The Task Architecture considered here, consists of

- 26 *Task*s connected with
- 69 *Signal*s.

Table A.1 (in the appendix) shows flash, RAM and safety level (ASIL) annotations of the *Task*s.

Hardware resources The *Hardware Resources* considered here, consist of

- 6 *ProcessingUnit*s where
  - 5 *ProcessingUnit* are concrete (CAM2, CAM1, FCU, Steering, Brake) and
  - 1 *ProcessingUnit* is variable (RAD) and contains 3 different variants (PU_RAD_A, PU_RAD_B, PU_RAD_C) where one variant is chosen during exploration,
- 4 *Bus*es (GSML, CAN_FD, CAN_FD_1, CAN_FD_2).

Figure 8.1 illustrates the "150% model" [84] of all *ProcessingUnit*s and their possible connections to different *Bus*es. For example, the FCU *ProcessingUnit* can be connected to any of the CAN_FD *Bus*es and to the GSML Bus. During the exploration the connections of the FCU *ProcessingUnit* are chosen entailing the following decisions:

- Is a connection to the GSML bus needed?
- Is a connection to one (and only one) of the CAN FD *Bus*es required?

In both cases, the exploration can also result in no connection between the FCU *ProcessingUnit* and the *Bus*es.

Table A.2 (in the appendix) shows flash, RAM, hardware cost, power consumption, safety level (ASIL) and weight annotations of the *Hardware Resource*s.

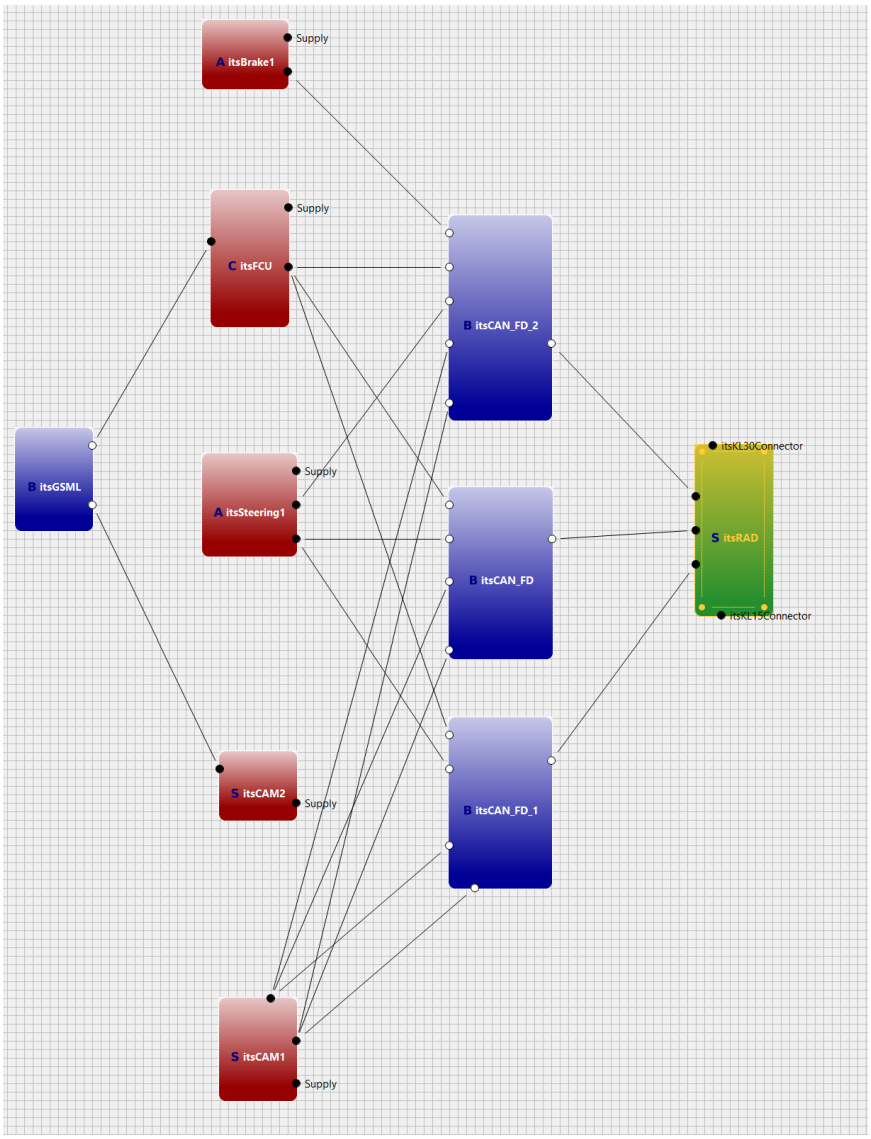


Figure 8.1.: 150% model of *Hardware Resources* including all possible connections between *ProcessingUnit*s (red: concrete units, green: variable unit) and *Bus*es (blue).

8.2.2.2. Language patterns

The Task Architecture, the Hardware Architecture (in terms of *Hardware Resources*) and the respective annotations are formalized using the DSML (cf. Chapter 6) in the form of the introduced basic patterns. Additionally, we consider the following language patterns:

- Constraints

  - Memory Pattern (see also 6.2.2.4) for
    - RAM (The sum of RAM of all *Task*s allocated to a ProcessingUnit may not exceed the RAM provided by that ProcessingUnit)
    - Flash memory (The sum of flash memory of all *Task*s allocated to a ProcessingUnit may not exceed the flash memory provided by that ProcessingUnit)
  - Safety Pattern (see also Section 6.2.2.3) (The ASIL of a *Task* allocated to a ProcessingUnit must be smaller or equal the ASIL of that ProcessingUnit)

- Optimization Objectives
  - Property Objective Pattern for (see also Section 6.2.3.1) (A certain *property* of *ProcessingUnit*s shall be minimized such that the resulting *Hardware Topology* is minimal considering that property)
    - Cost
    - Power consumption average
    - Weight
  - Bandwidth Objective Pattern (see also Section 6.2.3.3) (The used *bandwidth* ('used' by the *Signal*s which are sent via that *Bus*) of a given *Bus* shall be minimized). This objective pattern is used for all *Bus*es (CAN_FD, CAN_FD_1 CAN_FD_2, GSML).

8.2.2.3. E/E Architecture Exploration

In the following, the generated E/E Architecture solutions are described. As the validation phase was successful, and there are no contradicting constraints, we focus on the exploration in this section.

At first, we describe the different solutions according to their metrics. Then, we have a closer look at the the different *Hardware Topology* solutions in order to compare them, and show the trade off analysis, which has to be conducted by a system architect.

Table 8.2 illustrates the metrics for seven different calculated solutions. Six of the metrics where used as optimization objectives. The number of *ProcessingUnit*s (PUs) was not optimized. The table cells colored in green highlight optimal (minimal) values of each metric. A bandwidth value value of 0 means, that a *Bus* can be omitted. The table illustrates the pareto optimality of the different solutions. For instance, considering the power consumption, Solution_2 has a smaller and thus more optimal value compared to Solution_3. However, taking into account the cost and weight, Solution_3 is more optimal. This entails, that there is no solution which is optimal considering all objectives.

In order to choose a solution, a system architect has to consider the different *Hardware Topologies* of the solutions, too. Figure 8.2 shows the *Hardware Topology* of Solution_2 and Figure 8.3 of Solution_3. Solution_2, uses the Brake_PU which is connected to CAN_FD_2

	#PUs	Cost[€]	Power[W]	Weight[g]	Bandwidth [Byte/ms]			
					Can_FD	Can_FD_1	Can_FD_2	GSML
Solution_0	4	55	14	875	40	0	0	80015
Solution_1	4	55	14	875	42	0	0	80013
Solution_2	4	65	13	1025	0	0	42	80013
Solution_3	4	55	14	875	0	40	0	80015
Solution_4	4	55	14	875	0	42	0	80013
Solution_5	4	61	15	925	0	35	5	80140
Solution_6	4	75	22	1275	0	30	13	80013

Table 8.2.: Table describing the metrics of the E/E Architecture solutions (cells highlighted in green illustrate optimal values)

but does not use any of the Abstract_PU_RAD variants. Solution_3, in contrast, does not use the Brake_PU but uses PU_RAD_A. So the different metrics of those two solutions are a result of one *ProcessingUnit* which differs.

Taking into account Solution_5, which is shown in Figure 8.4, the topology of a solution may not only differ in the *ProcessingUnit*s which are used, but also in the communication of the the different *ProcessingUnit*s. Solution_5 uses three *Bus*es as opposed to Solution_2 and Solution_3 (2 *Bus*es). Furthermore, in Solution_5, PU_RAD_C was chosen which is connected via the additional CAN_FD_2 to PU_Steering. Considering the metrics, this solution is less optimal than Solutions_2 and Solution_3 in power consumption. It is less expensive than Solution_2, but more expensive than Solution_3 considering cost and weight. All solutions have in common, that the GSML *Bus* is used. This is due to the fact, that the data which is sent from the *Task Camera*, can only be sent via this *Bus* because of its size. Such a trade-off can only be solved by a system architect, who has to decide which solution can be chosen, due to the fact that he/she knows if, e.g., 1 more Watt of power is acceptable compared to €10 costs less.

The *Hardware Topologies* and Deployments of all solutions can be found in the appendix of this thesis (Chapter B).

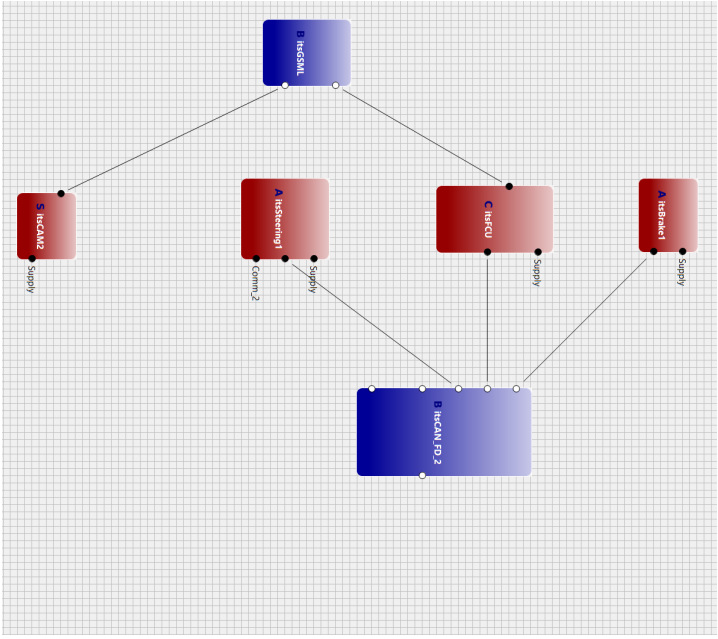


Figure 8.2.: Hardware Topology of Solution_2

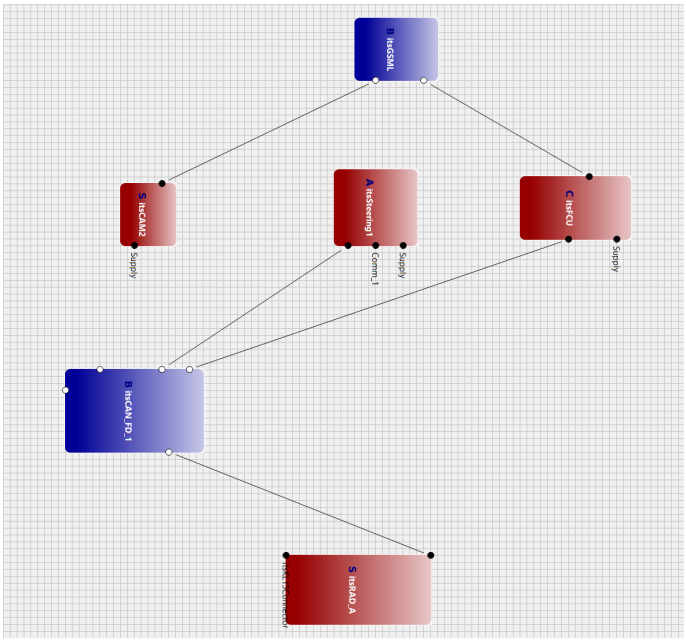


Figure 8.3.: Hardware Topology of Solution_3

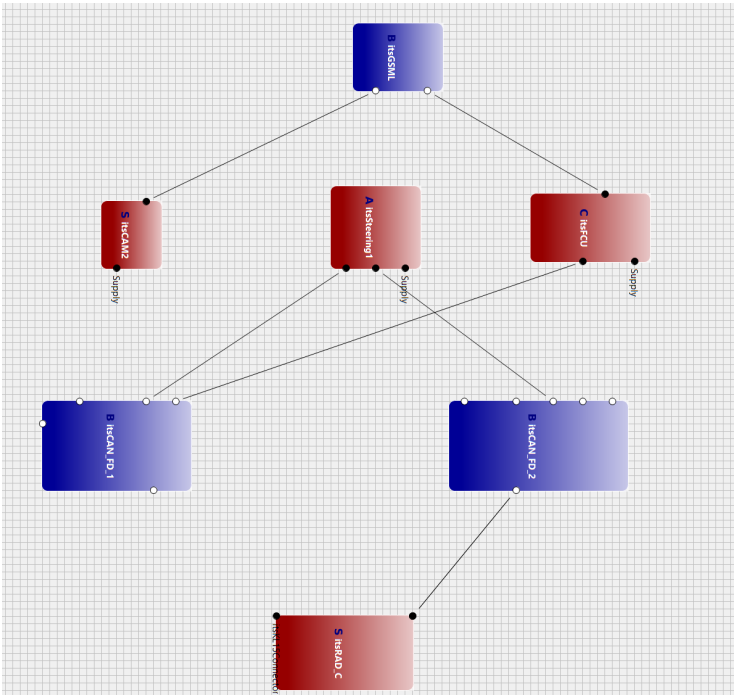


Figure 8.4.: Hardware Topology of Solution_5

9. Conclusion

In the following, we conclude this thesis by giving a short summary of all chapters in Section 9.1 and discuss the limitations of the presented approach in Section 9.2. In Section 9.4 we give an overall conclusion of this thesis and provide an outlook on future work in Section 9.3.

9.1. Summary

In Chapter 1, we described the motivation for this thesis which is the rising complexity of today's systems which makes their engineering more and more difficult. We thereby focused on 4 different aspects of complexity: software, hardware, integration and variability. Considering the engineering of automotive E/E Architectures, those architectures are consisting of more than 100 ECUs. As those distributed architectures are getting too complex and thus almost impossible to be built manually, which, most of all, involves adherence to ISO 26262. Due to this fact, the trend goes towards more centralized E/E Architectures, consisting of less but more powerful ECUs. Yet, the problem is how get from existing E/E Architectures to more centralized ones in the future. Hence, this thesis provides an approach to support a system architect in building future E/E Architectures, by providing dedicated models, a dedicated domain specific language, an exploration technique to find architectures and a methodology which integrates into an existing development approach.

In Chapter 2, we introduce the specifics of engineering automotive systems. First of all, due to the fact that safety plays an important role, this entails adherence to ISO26262, the safety standard for engineering E/E Architecture of passenger vehicles. It includes a dedicated model-based development process which is proposed by this standard. As model-based software engineering is, on the one hand, proposed by ISO 26262 and, on the other hand, promises to overcome the engineering complexity, we are introducing its most important aspects in this Chapter: viewpoints, components and interfaces. Lastly, we are also describing the general concept of automotive E/E Architectures, the current state of the architectures and future trends which focus on a minimization and centralization of E/E Architectures.

In Chapter 3, we elaborate on the related work of this thesis. There are four different areas which we are considering: architecture description languages (ADLs), domain specific languages (DSLs), Synthesis approaches (Design Space Exploration - DSE) and engineering

methodologies. Considering model-based ADLs, we are considering and comparing the UML and SysML, the UML Marte which is a real time extension of UML, AUTOSAR which is built for engineering automotive software systems, EAST ADL2 which provides a high level abstraction of AUTOSAR and the AADL which is built for engineering avionic systems. Regarding DSLs, there is one main language which is used in the modeling domain, the OCL. There are also more technology specific languages like SMTlib2 which are bound to a solving technique (SMT solvers in the case of SMTlib2). Considering synthesis or design space exploration approaches, we show that there has been done a lot of work in the area of solving the deployment problem. Furthermore there is also work which has been conducted in the area of Hardware or Topology synthesis (mostly for On-Chip-Networks). Lastly, there are also generic approaches independent from the specific problem or domain. Finally, we give an overview over engineering methodologies, focusing on two academic and two industrial approaches. SPES and CESAR are examples for academic approaches which have been developed in the course of research projects. IBM Harmony and AUTOSAR are industrial methodologies which haven been built for systems modeling with the SysML (IBM Harmony) and systems modeling with the AUTOSAR language. Thus, this Chapter covers the related work of the four main chapters, the three viewpoint Chapters E/E Architecture (ADLs), Specification (DSLs) and Exploration (DSE) and the methodology Chapter.

In Chapter 4, we are introducing the methodology proposed in this thesis. On the one hand, we are introducing the Model, Specification and Exploration Viewpoint and their connection with each other. Part of the models in the E/E Architecture Viewpoint are input to the Specification Viewpoint where those models, together with constraints and objectives are formalized using the domain specific language which is introduced in the Specification Viewpoint. This formalization is then passed to the Exploration Viewpoint where it is validated and explored. The validation is focusing on finding contradicting constraints which my give a hint about contradicting requirements which have to be resolved before continuing with the exploration. The exploration is then finding different *Hardware Topology* and deployment solutions which is fed back into the E/E Architecture Viewpoint. On the other hand, we show how our exploration integrates into the SPES system engineering methodology and how our methodology can be integrated into a customized V-Model XT development approach. We conclude by proposing a new role in the development process: the exploration engineer. The exploration engineer would support a system architect which is responsible for creating the system architecture (E/E Architecture Viewpoint) by providing the knowledge of how to automate the process of finding such architectures via formalization (Specification Viewpoint) and exploration (Exploration Viewpoint).

In Chapter 5, we are introducing the E/E Architecture Viewpoint providing all the necessary meta-models to create a model of an automotive E/E Architecture in early design phases. We thereby distinguish between models for Software and Hardware Architecture and the Deployment. The model of the Software Architecture which is considered in this

thesis is the Task Architecture consisting of *Task*s which execute a certain functionality and communicate with each other via *Signal*s (further software models like partitions or schedules are not considered in the thesis). Considering the models of the Hardware Architecture we distinguish between *Hardware Resources* and Hardware Topology. *Hardware Resources* are the computation (*ProcessingUnit*s/ ECUs) and communication resources (*Bus*es) for which different variants may exist. The *Hardware Topology* consists of *Hardware Resources* which are connected with each other and thus form the topology of an E/E Architecture. The deployment is connecting both software and Hardware Architecture. *Task*s are allocated to a computation resource in the *Hardware Topology* and *Signal*s are allocated to communication resources. Thus, an instance of an E/E Architecture can be described by a *Hardware Topology* and a *Task*/*Signal* deployment.

In Chapter 6, we are introducing the Specification Viewpoint providing a domain specific language capable of formalizing a E/E Architecture exploration problem. We therefore introduce a grammar defining how expressions in our language can be composed. Based on this grammar we are then defining language patterns which are inherent to an automotive E/E Architecture exploration problem definition. There are three types of patterns, basic patterns, constraint patterns and objective patterns. The basic patterns are necessary in order to formalize a E/E Architecture exploration problem. On the one hand, they formalize how a correct *Hardware Topology* can be built. On the other hand, they formalize the variability of *Hardware Resources* such that a correct variant of a resource is chosen. The constraint and objective patterns show formalized requirements which are typically encountered in the automotive domain and which we created during industrial collaborations with Continental. The constraint patterns (allocation/dislocation, function coupling/de-coupling, safety, memory) have to hold in order to form a valid and thus correct E/E Architecture. The objective patterns (property, cardinality, bandwidth) enable the optimization of an E/E Architecture into a certain direction e.g. the minimization of costs.

In Chapter 7, we are introducing the Exploration Viewpoint based on an SMT formulation in Z3, in order to solve the exploration problem which has been formalized in the Specification Viewpoint. We therefore provide a problem definition which is stored in a dedicated meta-model. As we chose to use SMT solving as solving technique in this thesis, we are providing a transformation of all patterns introduced in the Specification Viewpoint into SMT. Furthermore, we show how solutions are generated for both phases of validation and exploration. During the validation phase, we make use of the unsat core feature of the Z3 SMT solver which enables the calculation of contradicting constraints. When there are no contradicting constraints the exploration phase can be conducted. In this phase, different optimized solutions are calculated due to the formalized optimization objectives. We are furthermore providing metrics and visualization techniques in order to enable an efficient comparison of the different solutions.

In Chapter 8, we evaluate the work of this thesis by comparing the state of practice considering modeling an E/E Architecture in early phases of development, against our proposed exploration approach. In order to do so, we are defining a set of evaluation criteria which enable us to compare both approaches (Execution type of development steps, Execution duration of development steps, Process type support, Reaction speed to architectural changes, Optimization potential, Architecture verification type). Furthermore, we show the application of your approach to two industrial use cases which have been conducted during a collaboration with Continental. The first use case shows the application to calculate solutions for a deployment problem of *Task*s onto *ProcessingUnit*s and *Signal*s onto *Bus*es. The second use case shows the application of our approach to simultaneously calculate a *Hardware Topology* together with a deployment of *Task*s and *Signal*s, while also considering variability of *Hardware Resources*. Thus, this results in the calculation of a whole E/E Architecture.

In Chapter 9, we are concluding this thesis by giving a summary of all chapters, a conclusion of the underlying work by reconsidering the problem and contribution statement of this thesis and by giving an outlook on future work, entailing work which has not been dealt with in this thesis and possible extensions of this work.

9.2. Limitations

Before concluding this thesis, we discuss the limitations of the approach presented in this thesis. The allocation model presented in Chapter 5 allocates *Task*s to *ProcessingUnit*s. Thus, we assume a so called "bare-metal" execution of *Task*s on the processing units. This assumption does not consider that a *ProcessingUnit*s might have a real time operating system, providing a virtualization layer, that guarantees a timely execution of tasks. Such a guarantee might be necessary especially considering safety critical tasks like the calculation of an emergency brake. Furthermore, such an operating system would consume additional resources, e.g., RAM or Flash memory. This implies that less resources are actually available influencing the number of *Task*s that may be allocated to such a *ProcessingUnit*. (Limitation 1)

Another limitation of the presented approach is, that the timing of tasks is not considered and the timing of signals is only roughly estimated through the usage of the Bandwidth Objective Pattern. For example, the timing of two tasks exchanging information via a signal might have a certain end-to-end deadline which has to be met, e.g., due to the fact that a braking signal is transmitted. The Bandwidth Objective Pattern only takes care that an overload of a certain bus is prevented by minimizing its busload. (Limitation 2)

A third limitation of the presented approach is the solving technology that we use in the exploration viewpoint. A SMT solver provides an exact method to calculate the solution of a given problem, in our case, an E/E architecture exploration problem. An automotive subsystem like presented in [23] took 2h to calculate optimized solutions. Considering the

calculation of an E/E Architecture of a whole car, this time might rise up to days due to the nature of the exact method SMT. Such a high calculation time might be infeasible for a System Architect, who has to re-explore an architecture whenever something changes e.g. a new constraint has to be added or a processing unit has to be removed. (Limitation 3)

9.3. Future Work

Finally, we want to elaborate on possible future work of this thesis which can extend and improve the presented exploration approach. Considering the evaluation against the state of practice there are two criteria which could be improved: the execution type which is semi-automatic (see also Section 8.1.1) and the execution duration which is short to medium (see also Section 8.1.2). Considering possible extensions of our approach, they could be achieved by considering additional models in the E/E Architecture Viewpoint.

Considering the evaluation, the execution type of our approach is semi-automatic. This means that part of the work is automated (*Hardware Topology* and deployment calculation), whereas other parts are done manually. This includes the creation of the Task Architecture. Regarding the SPES methodology, this means answering the question of how to automatically get from the a logical architecture of a system created in the logical viewpoint, to the Task Architecture in the technical viewpoint. Due to the fact, that the logical architecture is independent from technical aspects of the system, an automatic calculation of the Task Architecture would have to cover this aspect, especially as logical components may also be realized by *Hardware Resources*. Among others, this entails the following questions: How can we automatically distinguish between logical components which can be realized as *Task*s as opposed to *Hardware Resources*? What are rules for the creation of a *Task* out of a set of logical components and how many logical components can be realized by a *Task*? Which of the properties of logical components have to be taken into account?

Taking into account the approach itself, it could be made even more precise by considering additional models in the exploration by extending the E/E Architecture Viewpoint. On the one hand, by considering a virtualization layer in the software architectures enabling partitioning (Limitation 1), on the other hand, considering the timing of *Task*s and *ProcessingUnit*s, resulting not only in a more precise solution but also in the calculation of respective schedules (Limitation 2).

A virtualization layer would enable the modeling of partitions which can be used to separate critical from non-critical parts of the software which can thus still be executed on the same *Hardware Resource*. In particular, considering more powerful *Hardware Resources*, like multi-core *ProcessingUnit*s, such a virtualization layer can ensure e.g. different safety integrity levels on the same *ProcessingUnit*. Regarding the trend towards more central E/E Architectures, such a mix of criticalities is necessary to ensure safety, if only few *ProcessingUnit*s build an E/E Architecture.

The consideration of timing would make our approach even more precise, due to the fact that, it can influence the *Allocation* of *Task*s to *ProcessingUnit*s and *Signal*s to *Bus*es.

It would extend the existing approach by additional calculation of schedules of *Task*s and *Signal*s which ensures that timing requirements are met in the calculated *Hardware Topology* and deployment.

The execution duration of our approach is small [h] to medium [d], depending on the size of the architectures. Considering industrial sized architectures, the execution duration is mostly medium which means that a calculation can last days (Limitation 3). This entails that the execution duration could still be improved. In the Specification Viewpoint, we presented a domain specific language capable of formalizing a E/E Architecture exploration problem. Due to the fact that this language is independent from the exploration technique (Exploration Viewpoint) which is used to solve the problem, other solving techniques could be used. In this thesis the Exploration Viewpoint is based on SMT but could be replaced by a different technology.

Inter Linear Programming (ILP) could be another exact approach which is based on a integer encoding of the problem as opposed to a Boolean encoding in SMT. Heuristic methods, like evolutionary algorithms could be promising to reduce the calculation duration, however at the cost of precision, as they are not exact methods and might not be able to calculate optimal solutions like an SMT approach. The most promising results could probably be achieved by a mix of different technologies covering the strengths of each method. Combining heuristics with an SMT approach, e.g., may overcome precision issues while at the same time speeding up the calculation.

9.4. Conclusion

In this thesis, we provide a holistic exploration approach enabling the automatic calculation of E/E Architectures to answer the overall problem asked in the introduction "How can we support a system architect during the development of future E/E Architectures?". In order to achieve this, we introduced a comprehensive approach, covering modeling aspects in the E/E Architecture Viewpoint, formalization aspects in the Specification Viewpoint and automatic calculation of solutions in the Exploration Viewpoint. Those three viewpoints provide the answers for research questions RQ1 (Which **models** are needed to precisely describe an E/E Architecture and how are they defined?), RQ2 (How to **formalize** E/E Architecture models and how to formalize requirements which have to be satisfied by an E/E Architecture?), and RQ3 (How can we formally define an E/E Architecture **exploration** problem which enables the calculation of valid and optimized or even optimal E/E Architectures taking into account deployment and variability aspects?). They are furthermore complemented by an exploration methodology which is integrated into an exemplary V-Modell XT development process showing how our approach can be seamlessly integrated into existing development processes, answering RQ4 (How does a dedicated exploration **methodology** for E/E Architectures look like and how does it integrate into an existing development process?).

The scientific contribution of this thesis therefore lies in the combined usage of techniques,

which, by themselves, are scientifically understood and have mostly been solved. However, a comprehensive approach taking into account the different aspects which are considered in thesis is missing. This also entails a dedicated exploration methodology which can be seamlessly integrated into an existing development process.

Through continuous collaborations with Continental [21, 23, 24] we could get a deep insight into industrial practice considering the modeling of E/E Architectures at early stages of development. We therefore see the need of a new role of an exploration engineer in the development process. A system architect, which is responsible for modeling an E/E Architecture has a deep knowledge and experience in building those architectures. He is thus able to create those architectures. However, this *Task* gets increasingly complex considering the rising complexity of E/E Architectures, regarding the intricate dependencies which are introduced by driver assistance and automated driving functions. We have observed a rising industrial need for automated support, as the process of building E/E Architectures is mostly based on experiences of few people and is also opinion driven as we already pointed out in [21]. So there is a need, not only for an automatized but also an objective approach which can calculate different E/E Architectures enabling a quantitative comparison.

Additionally, through this collaboration, we were able to evaluate our approach against the state of practice which shows that our approach can considerably improve the state of practice. Not only due to the fact that a manual process is automated and the duration of creating an E/E Architecture is thus reduced and no longer requires the manual work of a system architect. But also because the introduced approach allows for an agile process as opposed to a waterfall-like process. This is because each change can be covered by a new automatic exploration run and has not to be covered manually. Hence, a complex change process does not have to be triggered. Furthermore, each exploration also automatically takes care about the correctness of results which had to be checked manually before. Lastly, the possibility to optimize E/E Architectures can have a considerable impact as this was almost impossible before, due to the manual process. This means that an optimization can e.g. reduce the costs of an E/E Architecture which can have a big impact considering the mass production of vehicles.

A. E/E architecture evaluation model annotations

The following tables show the annotation values which were used for the evaluation of the E/E architecture exploration in Section 8.2.2.1.

Task\Annotation	Flash [kByte]	RAM [kByte]	ASIL
Acceleration_Pedal	5.000	2.000	B
Arbitrator	5.000	2.000	D
Brake_Pedal	5.000	2.000	D
Hydraulic_Modulator	5.000	2.000	B
LOC	1.000	1.000	B
ACC	81.000	3.600	D
ADAS_Monitoring	22.000	2.000	D
Boundary	10.000	4.000	C
Camera	30.000	120.000	B
Control Signal	10.000	4.000	B
DMC	10.000	4.000	B
Data Processing	10.000	10.000	D
Driver Monitoring	10.000	4.000	B
EBS	5.000	5.000	D
EPS	15.000	6.000	D
LRR	5.000	9.800	B
Mode Control	10.000	2.000	D
Safety	6.000	3.000	D
SituationAssessment	10.000	4.000	D
Trajectory	30.000	2.000	D

Table A.1.: Annotations of Tasks

HW Resource\Annotation	Flash [kB]	RAM [kB]	ASIL	Bandwidth [B/ms]
PU_Brake	40.000	30.000	D	-
PU_CAM1	30.000	120.000	C	-
PU_CAM2	40.000	30.000	D	-
PU_FCU	100.000	20.000	D	-
PU_Steering	40.000	20.000	D	-
Abstract_PU_RAD	-	-	-	-
- PU_RAD_A	20.000	5.000	D	-
- PU_RAD_B	40.000	15.000	C	-
- PU_RAD_C	50.000	20.000	B	-
Can_FD	-	-	-	1.500
Can_FD_1	-	-	-	1.500
Can_FD_2	-	-	-	1.500
GSML	-	-	-	200.000

Table A.2.: Annotations of ProcessingUnits and Buses

B. E/E architecture evaluation solutions

The following figures show each of the solutions described in the evaluation in Section 8.2.2. Each E/E architecture solution is described with the Hardware Topology and the respective Deployment.

	# PUs	Cost[€]	Power[W]	Weight[g]	Bandwidth [Byte/ms]			
					Can_FD	Can_FD_1	Can_FD_2	GSML
Solution_0	4	55	14	875	40	0	0	80015
Solution_1	4	55	14	875	42	0	0	80013
Solution_2	4	65	13	1025	0	0	42	80013
Solution_3	4	55	14	875	0	40	0	80015
Solution_4	4	55	14	875	0	42	0	80013
Solution_5	4	61	15	925	0	35	5	80140
Solution_6	4	75	22	1275	0	30	13	80013

Table B.1.: Table describing the metrics of the E/E architecture solutions (equal to table 8.2)

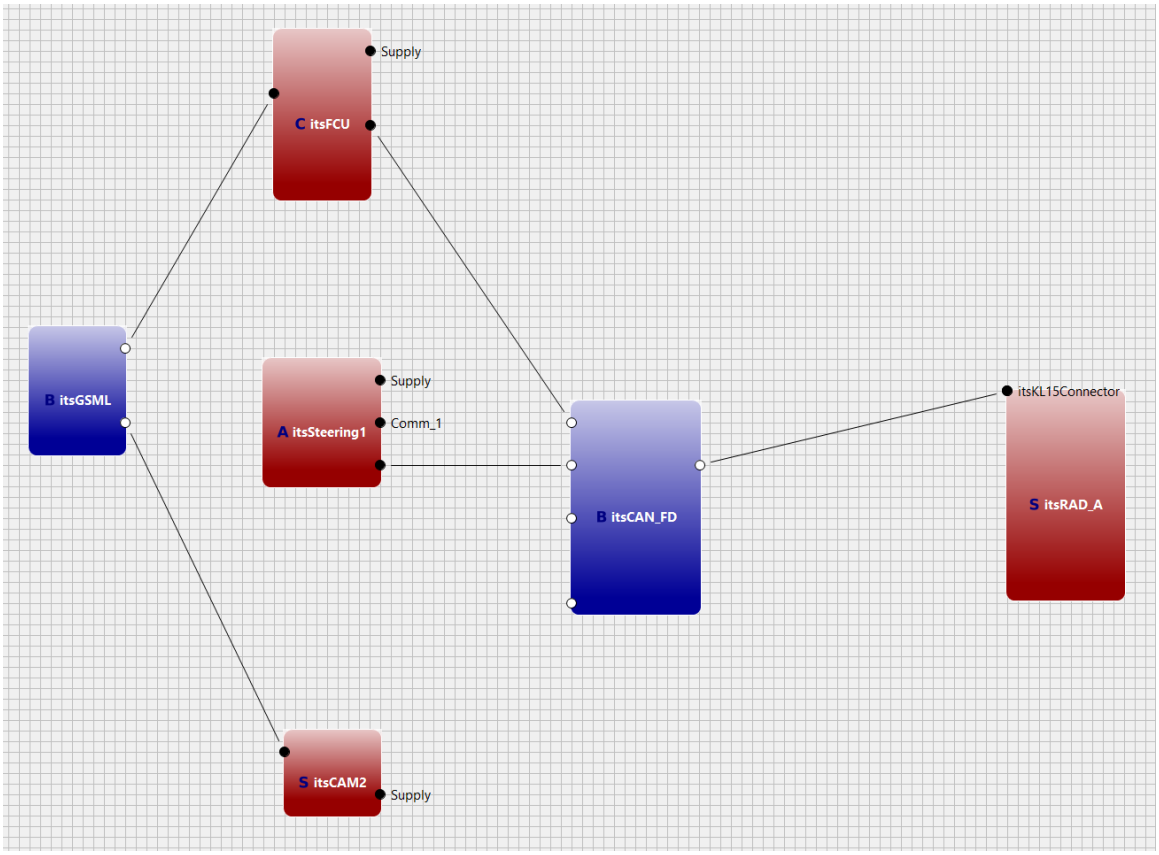


Figure B.1.: Hardware Topology of Solution_0

Task	Processing Unit
Acceleration_Pedal	PU_CAM2
Arbitrator	PU_Steering
Brake_Pedal	PU_RAD_A
Hydraulic_Modulator	PU_Steering
LOC	PU_Steering
ACC	PU_FCU
ADAS_Monitoring	PU_FCU
Boundary	PU_CAM2
Camera	PU_CAM2
Control Signal	PU_RAD_A
DMC	PU_Steering
Data Processing	PU_FCU
Driver Monitoring	PU_RAD_A
EBS	PU_Steering
EPS	PU_Steering
LRR	PU_FCU
Mode Control	PU_FCU
Safety	PU_Steering
SituationAssessment	PU_FCU
Trajectory	PU_Steering

Table B.2.: Deployment of Solution_0

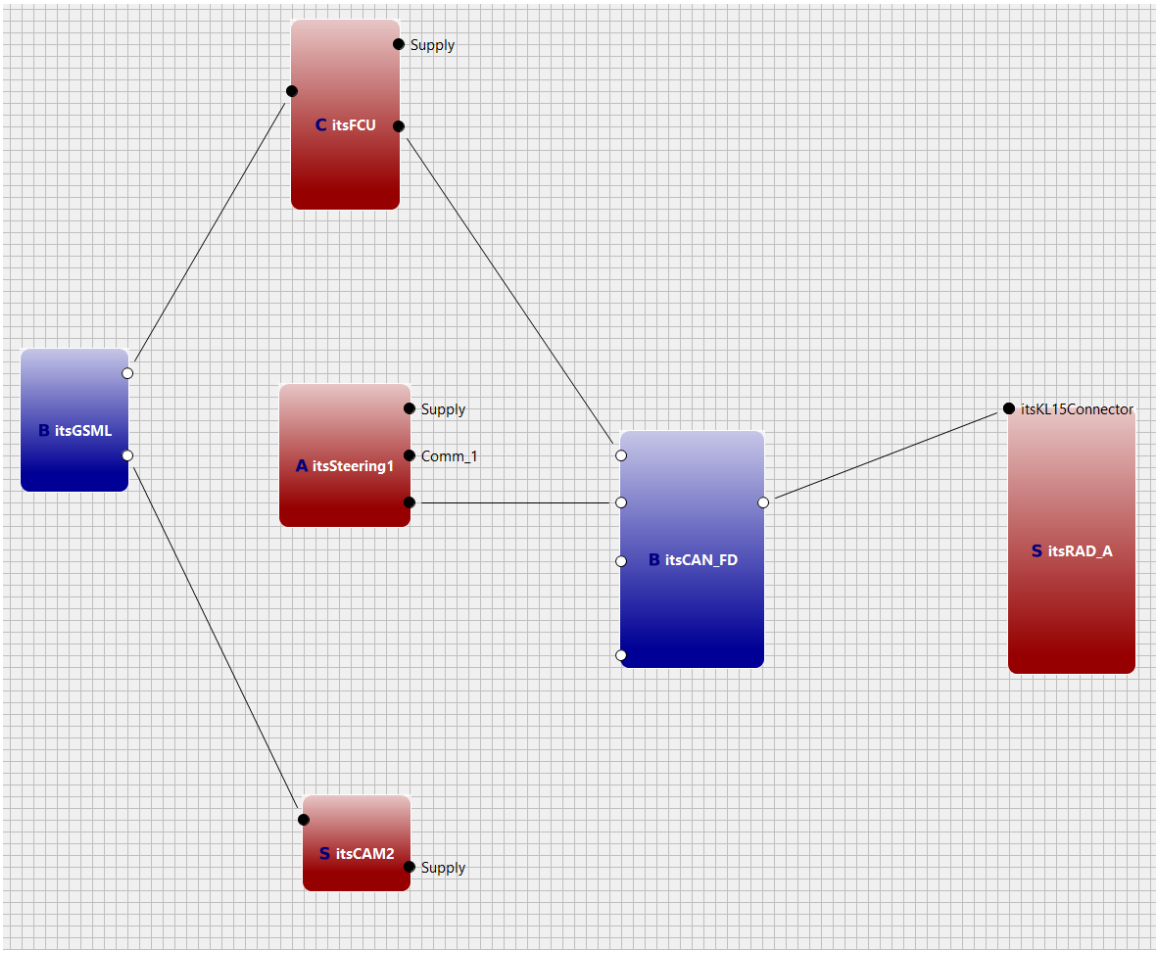


Figure B.2.: Hardware Topology of Solution_1

Task	Processing Unit
Acceleration_Pedal	PU_RAD_A
Arbitrator	PU_Steering
Brake_Pedal	PU_RAD_A
Hydraulic_Modulator	PU_Steering
LOC	PU_Steering
ACC	PU_FCU
ADAS_Monitoring	PU_FCU
Boundary	PU_CAM2
Camera	PU_CAM2
Control Signal	PU_RAD_A
DMC	PU_Steering
Data Processing	PU_FCU
Driver Monitoring	PU_RAD_A
EBS	PU_Steering
EPS	PU_Steering
LRR	PU_FCU
Mode Control	PU_FCU
Safety	PU_Steering
SituationAssessment	PU_FCU
Trajectory	PU_Steering

Table B.3.: Deployment of Solution_1

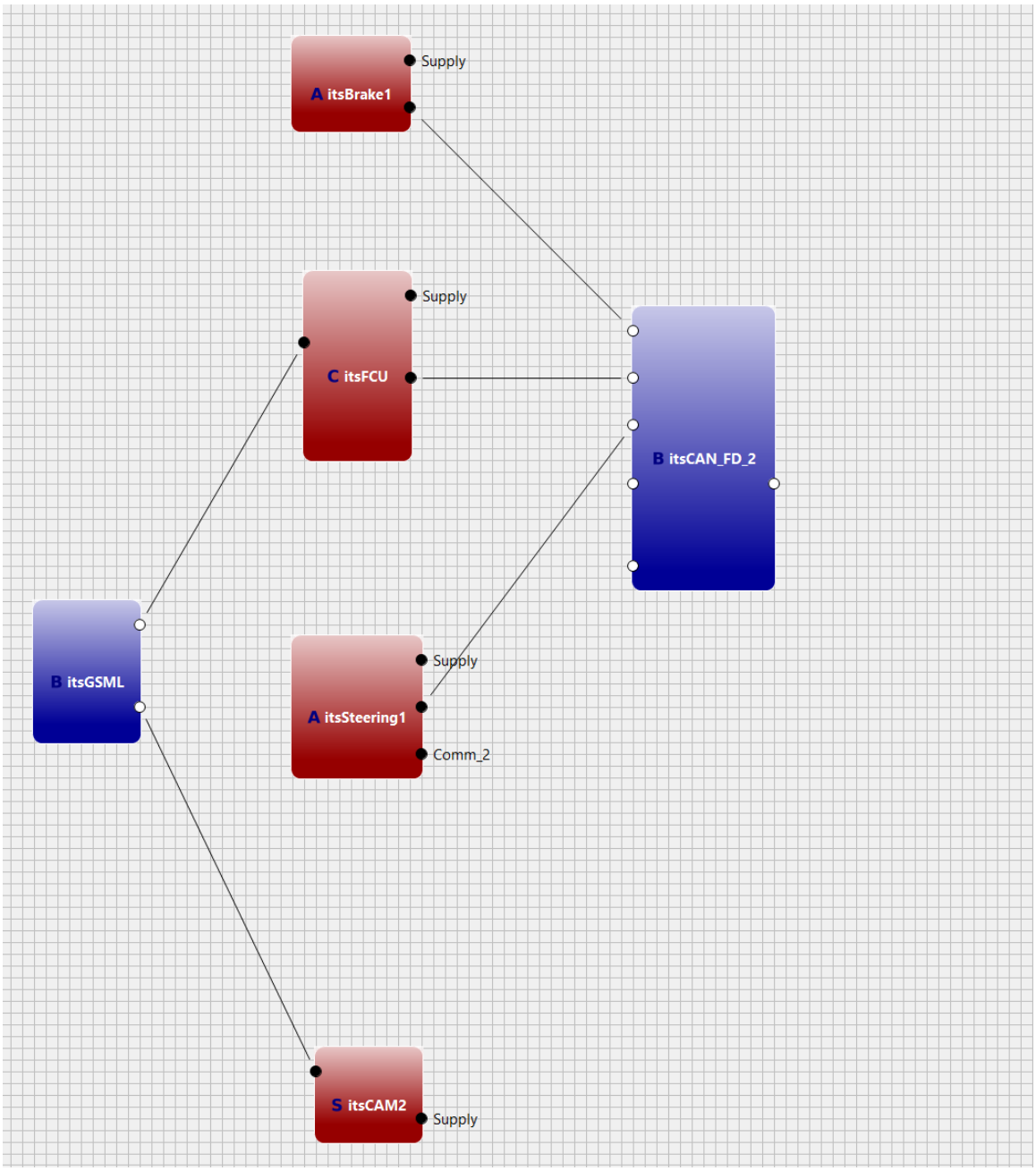


Figure B.3.: Hardware Topology of Solution_2

Task	Processing Unit
Acceleration_Pedal	PU_Brake
Arbitrator	PU_Steering
Brake_Pedal	PU_Brake
Hydraulic_Modulator	PU_Steering
LOC	PU_Steering
ACC	PU_FCU
ADAS_Monitoring	PU_FCU
Boundary	PU_CAM2
Camera	PU_CAM2
Control Signal	PU_Brake
DMC	PU_Steering
Data Processing	PU_FCU
Driver Monitoring	PU_Brake
EBS	PU_Steering
EPS	PU_Steering
LRR	PU_FCU
Mode Control	PU_FCU
Safety	PU_Steering
SituationAssessment	PU_FCU
Trajectory	PU_Steering

Table B.4.: Deployment of Solution_2

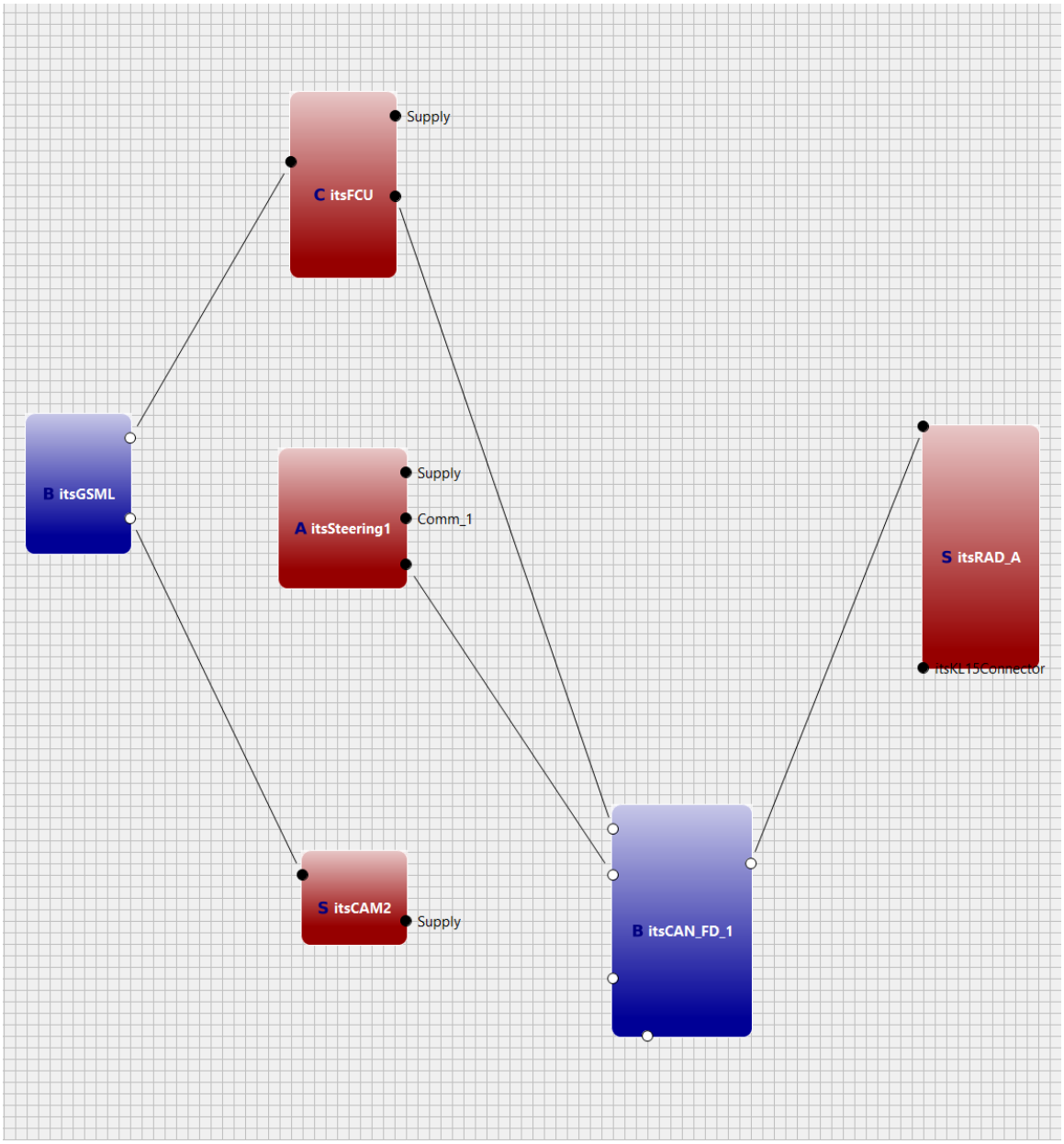


Figure B.4.: Hardware Topology of Solution_3

Task	Processing Unit
Acceleration_Pedal	PU_CAM2
Arbitrator	PU_Steering
Brake_Pedal	PU_RAD_A
Hydraulic_Modulator	PU_Steering
LOC	PU_Steering
ACC	PU_FCU
ADAS_Monitoring	PU_FCU
Boundary	PU_CAM2
Camera	PU_CAM2
Control Signal	PU_RAD_A
DMC	PU_Steering
Data Processing	PU_FCU
Driver Monitoring	PU_RAD_A
EBS	PU_Steering
EPS	PU_Steering
LRR	PU_FCU
Mode Control	PU_FCU
Safety	PU_Steering
SituationAssessment	PU_FCU
Trajectory	PU_Steering

Table B.5.: Deployment of Solution_3

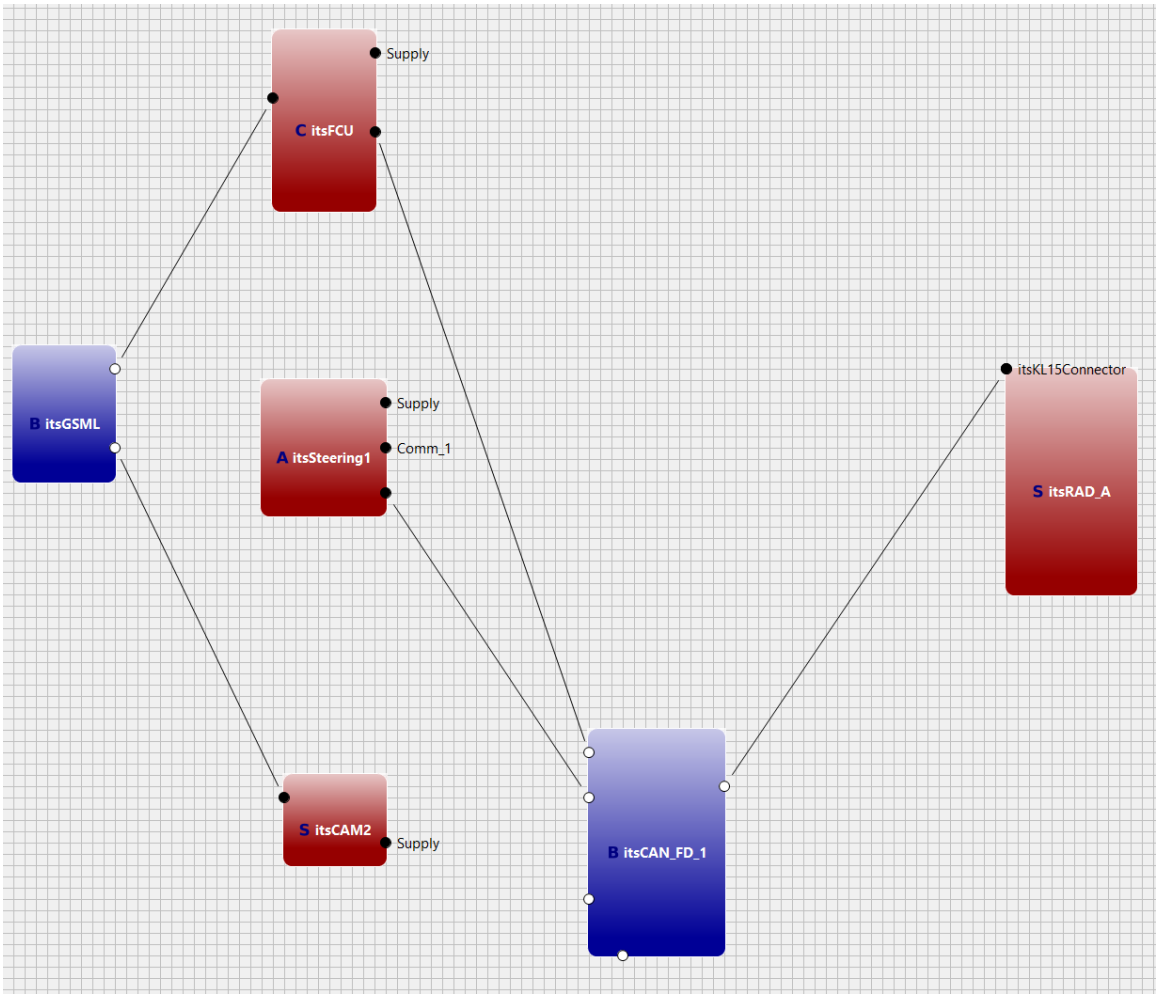


Figure B.5.: Hardware Topology of Solution_4

Task	Processing Unit
Acceleration_Pedal	PU_RAD_A
Arbitrator	PU_Steering
Brake_Pedal	PU_RAD_A
Hydraulic_Modulator	PU_Steering
LOC	PU_Steering
ACC	PU_FCU
ADAS_Monitoring	PU_FCU
Boundary	PU_CAM2
Camera	PU_CAM2
Control Signal	PU_RAD_A
DMC	PU_Steering
Data Processing	PU_FCU
Driver Monitoring	PU_RAD_A
EBS	PU_Steering
EPS	PU_Steering
LRR	PU_FCU
Mode Control	PU_FCU
Safety	PU_Steering
SituationAssessment	PU_FCU
Trajectory	PU_Steering

Table B.6.: Deployment of Solution_4

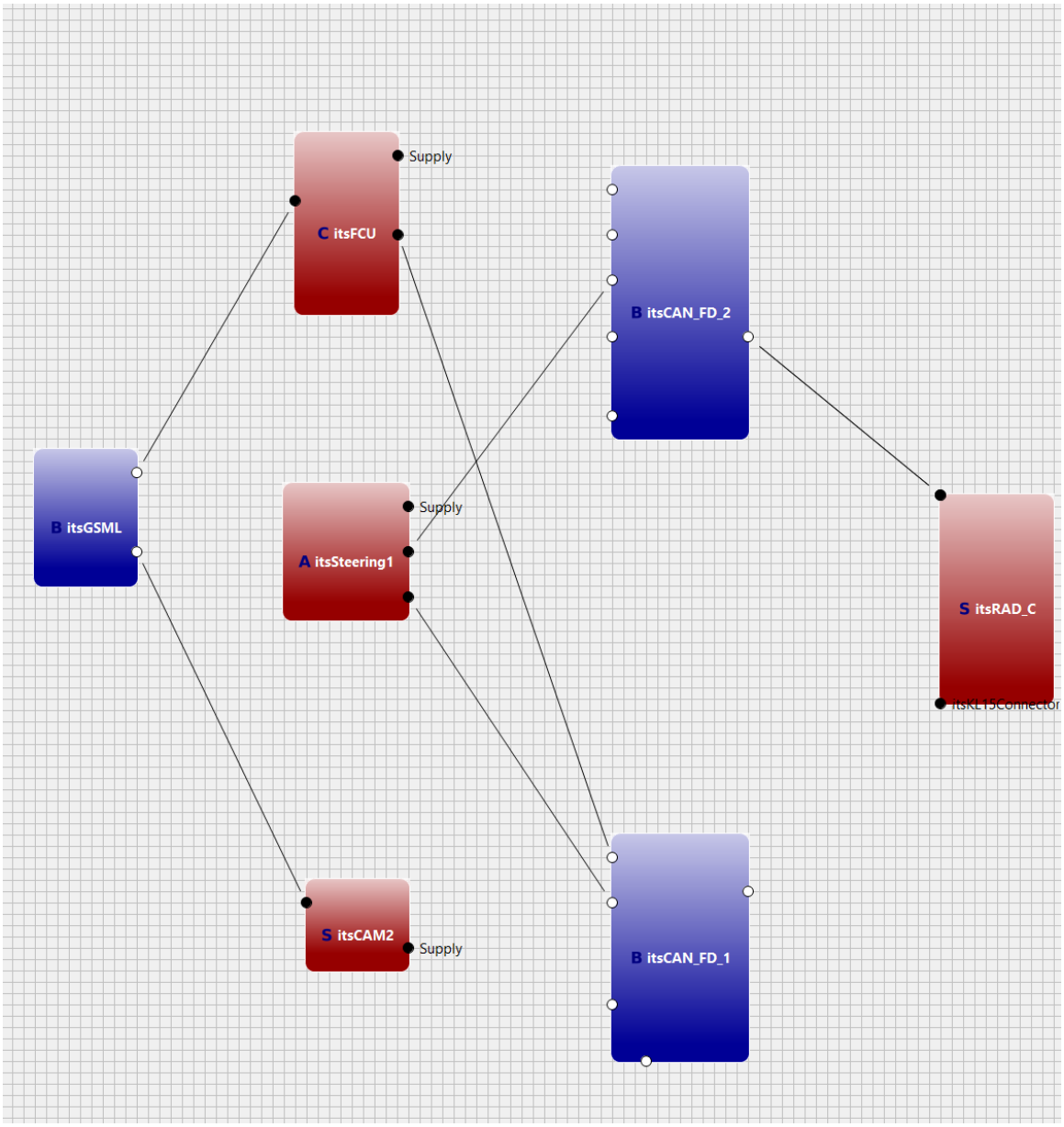


Figure B.6.: Hardware Topology of Solution_5

Task	Processing Unit
Acceleration_Pedal	PU_RAD_C
Arbitrator	PU_Steering
Brake_Pedal	PU_FCU
Hydraulic_Modulator	PU_Steering
LOC	PU_Steering
ACC	PU_FCU
ADAS_Monitoring	PU_FCU
Boundary	PU_CAM2
Camera	PU_CAM2
Control Signal	PU_RAD_C
DMC	PU_Steering
Data Processing	PU_FCU
Driver Monitoring	PU_RAD_C
EBS	PU_Steering
EPS	PU_Steering
LRR	PU_CAM2
Mode Control	PU_FCU
Safety	PU_Steering
SituationAssessment	PU_FCU
Trajectory	PU_Steering

Table B.7.: Deployment of Solution_5

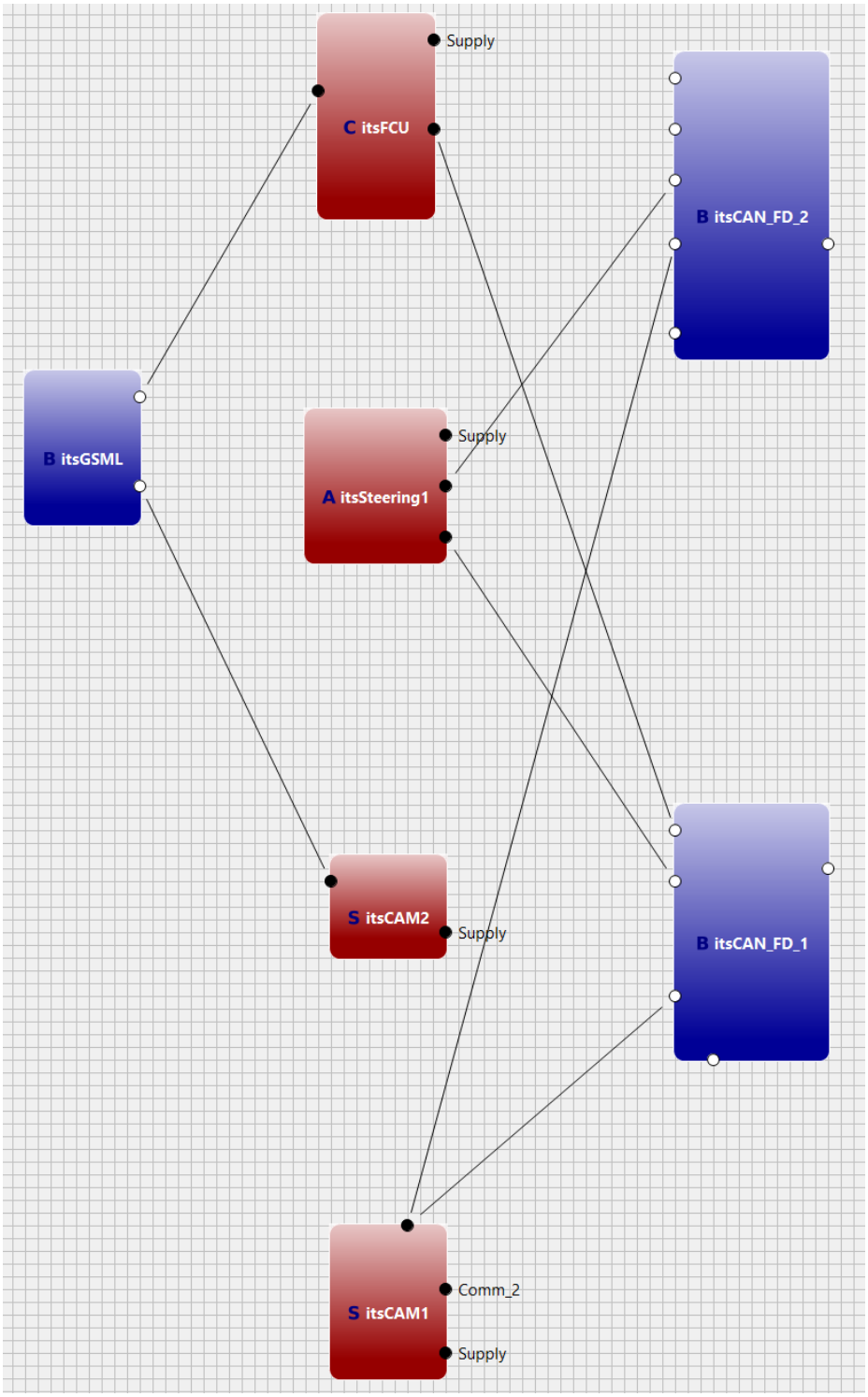


Figure B.7.: Hardware Topology of Solution_6

Task	Processing Unit
Acceleration_Pedal	PU_FCU
Arbitrator	PU_CAM1
Brake_Pedal	PU_FCU
Hydraulic_Modulator	PU_Steering
LOC	PU_Steering
ACC	PU_FCU
ADAS_Monitoring	PU_FCU
Boundary	PU_CAM2
Camera	PU_CAM2
Control Signal	PU_CAM1
DMC	PU_CAM1
Data Processing	PU_FCU
Driver Monitoring	PU_Steering
EBS	PU_Steering
EPS	PU_CAM1
LRR	PU_FCU
Mode Control	PU_CAM1
Safety	PU_Steering
SituationAssessment	PU_FCU
Trajectory	PU_CAM1

Table B.8.: Deployment of Solution_6

List of Figures

Bibliography

[1] Alexander Pretschner, Manfred Broy, Ingolf H. Krüger, and Thomas Stauner. Software engineering for automotive systems: A roadmap. FoSE 2007: Future of Software Engineering, pages 55–71, 2007.

[2] Marc Zeller and Christian Prehofer. Modeling and efficient solving of extra-functional properties for adaptation in networked embedded real-time systems. JOURNAL OF SYSTEM ARCHITECTURE, 2012.

[3] K. V. Prasad, M. Broy, and I. Krueger. Scanning advances in aerospace & automobile software technology. Proceedings of the IEEE, 98(4):510–514, April 2010.

[4] Stephan Sommer, Alexander Camek, Klaus Becker, Christian Buckl, Andreas Zirkler, Ludger Fiege, Michael Armbruster, Gernot Spiegelberg, and Alois Knoll. RACE : A Centralized Platform Computer Based Architecture for Automotive Applications. 2013.

[5] Tenny Benckendorff, Andreas Lapp, Thomas Oexner, and Thomas Thiel. Comparing current and future e/e architecture trends of commercial vehicles and passenger cars. In 19. Internationales Stuttgarter Symposium, pages 1190–1200. Springer, 2019.

[6] RTCA Special Committee 200. Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations. Rtca/Do-297, (2006):1–18, 2005.

[7] AEE Committee et al. Avionics application software standard interface: Arinc specification 653p1-2. Aeronautical Radio, 2007.

[8] OMG UML. 2.2 superstructure and infrastructure, february 2009. URL http://www.omg. org/spec/UML/2.2.

[9] Object Management Group. An OMG Systems Modeling Language TM Publication OMG Systems Modeling Language v1.5. 2017.

[10] Object Management Group. A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems (2008-06-08). 2(June), 2008.

[11] Simon Fürst, Jürgen Mössinger, Stefan Bunzel, Thomas Weber, Frank Kirschke-Biller, Peter Heitkämper, Gerulf Kinkelin, Kenji Nishikawa, and Klaus Lange. Autosar–a worldwide standard is on the road. In 14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden, volume 62, page 5, 2009.

[12] Philippe Cuenot, Patrick Frey, Rolf Johansson, Henrik Lönn, Yiannis Papadopoulos, Mark Oliver Reiser, Anders Sandberg, David Servat, Ramin Tavakoli Kolagari, Martin Törngren, and Matthias Weber. The EAST-ADL architecture description language for automotive embedded software. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 6100 LNCS:297–307, 2010.

[13] Peter H. Feiler, David P. Gluch, and John J. Hudak. The Architecture Analysis & Design Language (AADL): An Introduction. (February):1–145, 2006.

[14] Jos B Warmer and Anneke G Kleppe. The object constraint language: getting your models ready for MDA. Addison-Wesley Professional, 2003.

[15] Clark Barrett, Aaron Stump, Cesare Tinelli, et al. The smt-lib standard: Version 2.0. In Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England), volume 13, page 14, 2010.

[16] Klaus Pohl, Harald Hönninger, Reinhold Achatz, and Manfred Broy. Model-Based Engineering of Embedded Systems: The SPES 2020 Methodology. Springer Science & Business Media, 2012.

[17] Ajitha Rajan and Thomas Wahl. CESAR-cost-efficient methods and processes for safety-relevant embedded systems. Number 978-3709113868. Springer, 2013.

[18] Hans-Peter Hoffmann. IBM Rational Harmony Deskbook. 2014.

[19] AUTOSAR Administration. AUTOSAR_TR_Methodology. pages 1–520, 2017.

[20] Johannes Eder and Sebastian Voss. Usable design space exploration in autofocus3. In EduSymp/OSS4MDE@ MoDELS, pages 51–58, 2016.

[21] Johannes Eder, Sergey Zverlov, Sebastian Voss, Maged Khalil, and Alexandru Ipatiov. Bringing DSE to life: exploring the design space of an industrial automotive use case. In 2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS), 2017.

[22] Johannes Eder. Exploration of hardware topologies based on functions, variability and timing. In Proceedings of the 21st ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, pages 145–149, 2018.

[23] Johannes Eder, Andreas Bayha, Sebastian Voss, Alexandru Ipatiov, and Maged Khalil. From deployment to platform exploration - automatic synthesis of distributed automotive hardware architectures. In 2018 ACM/IEEE 21th International Conference on Model Driven Engineering Languages and Systems (MODELS), 2018.

[24] Johannes Eder, Sebastian Voss, Andreas Bayha, Alexandru Ipatiov, and Maged Khalil. Hardware architecture exploration: automatic exploration of distributed automotive hardware architectures. Software & Systems Modeling, 19(4), 2020.

[25] ISO 26262 - Road vehicles - Functional safety.

[26] IE Commission et al. Iec 61508: Functional safety of electrical/electronic/programmable electronic safety related systems, 2010.

[27] Laura E Hart. Introduction to model-based system engineering (mbse) and sysml. In Delaware Valley INCOSE Chapter Meeting. Ramblewood Country Club Mount Laurel, New Jersey, 2015.

[28] Jeff Bergenthal. Final report model based engineering (mbe) subcommittee. National Defense Industrial Association, Arlington, VA, 2011.

[29] Christel Baier and Joost-Pieter Katoen. Principles of model checking. MIT press, 2008.

[30] ISO/IEC 42010 Systems and Software Engineering — Architectural Description.

[31] ISO ISO. Iec 10746-1 information technology–open distributed processing–reference model: Overview, 1998.

[32] P Cuenot, P Frey, R Johansson, H Lönn, M Reiser, D Servat, R Tavakoli Kolagari, and D J Chen. Developing Automotive Products Using the EAST-ADL2 , an AUTOSAR Compliant Architecture Description Language. Architecture, 793(1-3):1–10, 2008.

[33] EA Lee, Haiyang Zheng, and Ye Zhou. Causality interfaces and compositional causality analysis. Foundations of Interface Technologies (FIT), Satellite to CONCUR, San Francisco, CA, pages 1–16, 2005.

[34] Manfred Broy and Ketil Stølen. Specification and development of interactive systems: focus on streams, interfaces, and refinement. Springer Science & Business Media, 2012.

[35] Manfred Broy and Gheorghe Ştefănescu. The algebra of stream processing functions. Theoretical Computer Science, 258(1-2):99–129, 2001.

[36] Steve V Hatch. Computerized engine controls. Cengage Learning, 2020.

[37] Manfred Broy. Challenges in automotive software engineering. Proceedings - International Conference on Software Engineering, 2006:33–42, 2006.

[38] Detlef Zerfowski and Andreas Lock. Functional architecture and e/e-architecture–a challenge for the automotive industry. In 19. Internationales Stuttgarter Symposium, pages 909–920. Springer, 2019.

[39] Morteza Hashemi Farzaneh Stefan Kugele, Vadim Cebotari, Mario Gleirscher, Diego Christoph Segler, Sina Shafaei, Hans-Jörg Vögel, Fridolin Bauer, Alois Knoll, and Hans-Ulrich Michel Marmsoler. Research Challenges for a Future-Proof E/E Architecture. Informatik 2017, pages 1463–1474, 2017.

[40] Matthias Traub, Alexander Maier, and Kai L. Barbehon. Future Automotive Architecture and the Impact of IT Trends. IEEE Software, 34(3):27–32, 2017.

[41] Martin Buechel, Jelena Frtunikj, Klaus Becker, Stephan Sommer, Christian Buckl, Michael Armbruster, Andre Marek, Andreas Zirkler, Cornel Klein, and Alois Knoll. An Automated Electric Vehicle Prototype Showing New Trends in Automotive Architectures. IEEE Conference on Intelligent Transportation Systems, Proceedings, ITSC, 2015-October:1274–1279, 2015.

[42] Lisa Braun, Michael Armbruster, and Eric Sax. Stakeholder issues concerning the automotive E/E-architecture. 2016 International Conference on Electrical Systems for Aircraft, Railway, Ship Propulsion and Road Vehicles and International Transportation Electrification Conference, ESARS-ITEC 2016, pages 1–6, 2016.

[43] Sebastien Gerard and Bran Selic. The UML - MARTE standardized profile. IFAC Proceedings Volumes (IFAC-PapersOnline), 17(1 PART 1):6909–6913, 2008.

[44] AUTOSAR. Layered Software Architecture V4.2. page 165, 2014.

[45] Anders Sandberg, Dejiu Chen, Henrik Lönn, Rolf Johansson, Lei Feng, Martin Törngren, Sandra Torchiaro, Ramin Tavakoli-Kolagari, and Andreas Abele. Model-based safety engineering of interdependent functions in automotive vehicles using EAST-ADL2. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 6351 LNCS:332–346, 2010.

[46] Jordi Cabot and Martin Gogolla. Object constraint language (OCL): A definitive guide. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 7320 LNCS:58–90, 2012.

[47] Clark Barrett, Roberto Sebastiani, Sanjit A Seshia, Cesare Tinelli, A Biere, M Heule, H van Maaren, and T Walsh. Handbook of satisfiability. Satisfiability modulo theories, 185:825–885, 2009.

[48] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard Version 2 . 5. 2015.

[49] Nikolaj Bjørner, Anh Dung Phan, and Lars Fleckenstein. νZ-an optimizing SMT solver. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 9035:194–199, 2015.

[50] Sebastian Voss and Bernhard Schatz. Deployment and scheduling synthesis for mixed-critical shared-memory applications. Proceedings of the International Symposium and Workshop on Engineering of Computer Based Systems, (April):100–109, 2013.

[51] Bernhard Schätz, Sebastian Voss, and Sergey Zverlov. Automating Design-space Exploration: Optimal Deployment of Automotive SW-components in an ISO26262 Context. Proceedings of the 52Nd Annual Design Automation Conference, pages 99:1—-99:6, 2015.

[52] Bernhard Schätz, Florian Hölzl, and Torbjörn Lundkvist. Design-Space Exploration through Constraint-Based Model-Transformation. 2010 17th IEEE International Conference and Workshops on Engineering of Computer Based Systems, pages 173–182, 2010.

[53] Stefan Kugele, Gheorghe Pucea, Ramona Popa, Laurent Dieudonne, and Horst Eckardt. On the deployment problem of embedded systems. 2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE 2015, pages 158–167, 2015.

[54] K. Becker and S. Voss. Analyzing graceful degradation for mixed critical fault-tolerant real-time systems. In 2015 IEEE 18th International Symposium on Real-Time Distributed Computing, pages 110–118, April 2015.

[55] Jordan A. Ross, Alexandr Murashkin, Jia Hui Liang, Michal Antkiewicz, and Krzysztof Czarnecki. Synthesis and exploration of multi-level, multi-perspective architectures of automotive embedded systems. Software and Systems Modeling, pages 1–29, 2017.

[56] Aldeida Aleti. Designing automotive embedded systems with adaptive genetic algorithms. Automated Software Engineering, 22(2):199–240, 2015.

[57] Ernest Wozniak, Asma Mehiaoui, Chokri Mraidha, Sara Tucci-Piergiovanni, and Sebastien Gerard. An optimization approach for the synthesis of AUTOSAR architectures. IEEE International Conference on Emerging Technologies and Factory Automation, ETFA, 2013.

[58] Gabriel Campeanu, Jan Carlson, and Severine Sentilles. Component allocation optimization for heterogeneous CPU-GPU embedded systems. Proceedings - 40th Euromicro Conference Series on Software Engineering and Advanced Applications, SEAA 2014, pages 229–236, 2014.

[59] Gabriel Campeanu and Jan Carlson. Allocation Optimization for Component-based Embedded Systems with GPUs. 2018.

[60] Patrick Leserf, Pierre De Saqui-Sannes, Jérôme Hugues, and Khaled Chaaban. Architecture optimization with sysML modeling: A case study using variability. Communications in Computer and Information Science, 580(February):311–327, 2015.

[61] Martin Lukasiewycz, Florian Sagstetter, and Sebastian Steinhorst. Efficient design space exploration of embedded platforms. Dac, pages 126–127, 2015.

[62] Alexander Metzner and Christian Herde. RTSAT - An optimal and efficient approach to the task allocation problem in distributed architectures. Proceedings - Real-Time Systems Symposium, pages 147–156, 2006.

[63] Twan Basten, Emiel Van Benthum, Marc Geilen, Martijn Hendriks, Fred Houben, Georgeta Igna, Frans Reckers, Sebastian De Smet, Lou Somers, Egbert Teeselink, Nikola Trčka, Frits Vaandrager, Jacques Verriet, Marc Voorhoeve, and Yang Yang. Model-driven design-space exploration for embedded systems: The octopus toolset. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 6415 LNCS(PART 1):90–105, 2010.

[64] Felice Balarin, Paolo Giusto, Attila Jurecska, Michael Chiodo, Claudio Passerone, Ellen Sentovich, Harry Hsieh, Luciano Lavagno, Bassam Tabbara, Alberto Sangiovanni-Vincentelli, et al. Hardware-software co-design of embedded systems: the POLIS approach. Springer Science & Business Media, 1997.

[65] Wei Peng, Hong Li, Min Yao, and Zheng Sun. Deployment optimization for AUTOSAR system configuration. ICCET 2010 - 2010 International Conference on Computer Engineering and Technology, Proceedings, 4:V4–189–V4–193, 2010.

[66] S. Graf, M. Glass, J. Teich, and C. Lauer. Multi-variant-based design space exploration for automotive embedded systems. Design, Automation & Test in Europe Conference & Exhibition (DATE), 2014, pages 1–6, 2014.

[67] Martin Lukasiewycz, Michael Glaß, Christian Haubelt, and Jürgen Teich. Efficient symbolic multi-objective design space exploration. Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC, pages 691–696, 2008.

[68] Dhananjay Thiruvady, I. Moser, Aldeida Aleti, and Asef Nazari. Constraint programming and ant colony system for the component deployment problem. Procedia Computer Science, 29:1937–1947, 2014.

[69] Shin-Haeng Kang, Hoeseok Yang, Sungchan Kim, Iuliana Bacivarov, Soonhoi Ha, and Lothar Thiele. Reliability-aware mapping optimization of multi-core systems with mixed-criticality. Design, Automation & Test in Europe Conference & Exhibition (DATE), 2014, pages 1–4, 2014.

[70] Uwe Pohlmann, Matthias Meyer, Andreas Dann, and Christopher Brink. Viewpoints and Views in Hardware Platform Modeling for Safe Deployment. Proceedings of the 2nd Workshop on View-Based, Aspect-Oriented and Orthographic Software Modelling - VAO '14, pages 23–30, 2014.

[71] Nikunj Bajaj, Pierluigi Nuzzo, Michael Masin, and Alberto Sangiovanni-Vincentelli. Optimized Selection of Reliable and Cost-Effective Cyber-Physical System Architectures. Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015, pages 561–566, 2015.

[72] Michael Glaß, Martin Lukasiewycz, Rolf Wanka, Christian Haubelt, and Jürgen Teich. Multi-objective routing and topology optimization in networked embedded systems. Proceedings - 2008 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, IC-SAMOS 2008, pages 74–81, 2008.

[73] Sudeep Pasricha and Nikil Dutt. Floorplan-aware automated synthesis of bus-based communication architectures. Proceedings of the 42nd Annual Design Automation Conference, pages 565–570, 2005.

[74] A Pinto, A Bonivento, R Passerone, and A Sangiovanni-Vincetelli. System level design paradigms: Platform-based design and communication synthesis. ACM Transactions on Design Automation of Electronic Systems, 11(3):537–563, 2006.

[75] S. Zverlov and S. Voss. Synthesis of pareto efficient technical architectures for multicore systems. In 2014 IEEE 38th International Computer Software and Applications Conference Workshops, pages 366–371, July 2014.

[76] Paolo Mantovani, Giuseppe Di Guglielmo, and Luca P. Carloni. High-level synthesis of accelerators in embedded scalable platforms. Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC, 25-28-Janu:204–211, 2016.

[77] Steffen Peter and Tony Givargis. Component-Based Synthesis of Embedded Systems Using Satisfiability Modulo Theories. ACM Transactions on Design Automation of Electronic Systems, 20(4):1–27, 2015.

[78] Ethan K. Jackson and Janos Sztipanovits. Towards a formal foundation for domain specific modeling languages. IEEE International Conference on Embedded Software, EMSOFT 2006, pages 53–62, 2006.

[79] E.K. Jackson, Eunsuk Kang, Markus Dahlweid, Dirk Seifert, and T. Santen. Components, platforms and possibilities: towards generic automation for MDA. Proceedings of the tenth ACM international conference on Embedded software, pages 39–48, 2010.

[80] Eunsuk Kang, Ethan Jackson, and Wolfram Schulte. An approach for effective design space exploration. Foundations of Computer Software. Modeling, Development, and Verification of Adaptive Systems, pages 33–54, 2010.

[81] Klaus Pohl, Manfred Broy, Heinrich Daembkes, and Harald Hönninger. Advanced model-based engineering of embedded systems. In Advanced Model-Based Engineering of Embedded Systems, pages 3–9. Springer, 2016.

[82] Manfred Broy and Andreas Rausch. Das neue V-Modell®1 XT – Ein anpassbares Vorgehensmodell für Software und System Engineering. pages 1–8.

[83] Klaus Pohl, Günter Böckle, and Frank J van Der Linden. Software product line engineering: foundations, principles and techniques. Springer Science & Business Media, 2005.

[84] Damir Bilic, Etienne Brosse, Andrey Sadovykh, Dragos Truscan, Hugo Bruneliere, and Uwe Ryssel. An Integrated Model-based Tool Chain for Managing Variability in Complex System Design. Models and Evolution Workshop (ME 2019), co-located with the IEEE / ACM 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS 2019), 2019.

[85] Andreas Bayha, Levi Lúcio, Vincent Aravantinos, Kenji Miyamoto, and Georgeta Igna. Factory product lines: Tackling the compatibility problem. In Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems, pages 57–64. ACM, 2016.

[86] Shahar Maoz, Ferdinand Mehlan, Jan Oliver Ringert, Bernhard Rumpe, and Michael von Wenckstern. Ocl framework to verify extra-functional properties in component and connector models. In MODELS (Satellite Events), pages 24–30, 2017.

[87] Sebastian Voss Alexander Diewald and Simon Barner. A Lightweight Design Space Exploration And Optimization Language. Acm, page 4503, 2015.