



Avoidance of inconsistencies during the virtual integration of vehicle software

(Based on the diploma thesis of Benjamin Honke)

Benjamin Honke

**Institut für Software & Systems Engineering
Universität Augsburg**

Report 2012-08

August 2012

INSTITUT FÜR INFORMATIK
D-86135 AUGSBURG

Copyright © Benjamin Honke
Institut für Informatik
Universität Augsburg
D-86135 Augsburg, Germany
<http://www.Informatik.Uni-Augsburg.DE>
— all rights reserved —

Abstract

In today's vehicles multiple programmable Electronic Control Units (ECU) are used in order to realize functional requirements of automotive manufacturers or Original Equipment Manufacturer (OEM). A vehicle of the luxury class contains more than 80 of such ECUs, which are networked with each other. Software development for these ECUs is being distributed effectively across several enterprises and countries. During the integration of developed software with hardware, inconsistencies do emerge constantly - the software does not work at a first attempt. Cost-intensive fault diagnostics and reworking result out of it. As way out, increasingly methods are being developed, which anticipate integration "virtually", on model level. Virtual integration represents anticipation of integration task on the left side, i.e. the development side, of the V-Cycle development process. Thereby integration should already be realized, for example, during the design phase on the model level. Thus interface compatibilities as well as a later performance within the embedded system can be verified at an early stage of development. For this reason, this thesis analyzes newer modeling and simulation approaches in regard to their application for automotive embedded systems in order to enable virtual integration and to simplify real integration.

Contents

1. Introduction	10
1.1. Motivation	10
1.2. Problems and Challenges	11
1.3. Outline	12
2. Background and Basics	14
2.1. AUTOSAR	14
2.1.1. AUTOSAR Architecture	15
2.1.2. Meta Model and Templates	18
2.1.3. Methodology	20
2.2. Architecture Description Languages	21
2.2.1. ADL Concept	21
2.2.2. EAST-ADL2	24
2.2.3. EAST-ADL2 vs. SEA-AADL and other approaches	29
2.3. Integration	31
2.3.1. Integration in general	31
2.3.2. Integration in an automotive environment	33
2.3.3. Special kinds of integration concerning models and views	38
2.3.4. AUTOSAR integration from a practical point of view	43
2.4. Inconsistency	45
2.4.1. Reasons for inconsistencies: Basic Conditions of Development	46
2.4.2. Categories of Inconsistencies between Software Components	47
2.4.3. Consequences of inconsistencies for integration	49
2.5. Related Work	51
2.5.1. CCI	51
2.5.2. UnSCom	52
2.5.3. Other integration platforms	53

3. Discovering mistakes of integration in practice	55
3.1. A Survey to find out current problems	55
3.1.1. Structure and goals of the survey	56
3.1.2. Interpretation of the surveys' answers	57
3.2. Results of the whole survey and further course	69
3.2.1. Reasons for inconsistencies	70
3.2.2. Concrete inconsistencies	72
4. Solving the actual problems using AUTOSAR, EAST-ADL2 and additional concepts	73
4.1. Analysis of the surveys' results	73
4.1.1. Why are models better than textual documents or executable code?	74
4.1.2. Basic Conditions & AUTOSAR	78
4.1.3. Concrete Inconsistencies & AUTOSAR	97
4.1.4. Basic Conditions & EAST-ADL2.0	116
4.1.5. Concrete Inconsistencies & EAST-ADL2.0	130
4.1.6. Conclusion	136
4.2. Extension of AUTOSAR and EAST-ADL2 by Semantics	138
4.2.1. Separation UnSCom from AUTOSAR and EAST-ADL2	140
4.2.2. Demonstration and Concretion of AUTOSARs' and EAST-ADL2s' semantical drawback	144
4.2.3. Linguistic Meta Model and Benefits	146
4.2.4. Application & Case Study	155
5. Conclusion and Outlook	158
5.1. Summary	158
5.2. Outlook	160
A. Acronyms	162
B. Survey	164
C. Analysis of the survey	173
D. Summary: Analysis' Results	182
E. Case Study: Memory Stack	184
E.1. Statement Collection: textual representation	184
E.1.1. Information Objects	185

Contents

E.1.2. Functions	185
E.1.3. Process	185
E.2. Statement Collection: graphical representation	185
F. Bibliography	190

List of Figures

1.1. Interconnected electronic control units in a modern vehicle [5]	10
2.1. AUTOSAR refined layered software architecture from [16]	16
2.2. Basic Autosar Approach from[16]	17
2.3. AUTOSAR Templates: Overview from [10]	18
2.4. Autosar Methodology: Overview from [16]	20
2.5. EAST-ADL2 Abstraction Layers and Artifacts from [9]	25
2.6. The seven artifacts of EAST-ADL2 and their interactions: Overview from [37]	26
2.7. A V-model for automotive development from [7]	28
2.8. AUTOSAR Interfaces from [16]	36
2.9. Model Integration (Schematical)	40
2.10.EAST-ADL2.0 Example of a mapping between Design and Implementation Level, from [9]	41
2.11.Inconsistencies Overview	50
4.1. AUTOSAR Components-Ports-Interfaces, from [12]	75
4.2. AUTOSAR Elements, from [12]	79
4.3. AUTOSAR Software component implementation, from [12]	80
4.4. AUTOSAR Model Persistence Rules for XML , from [14]	86
4.5. AUTOSAR Top level structure, from [12]	89
4.6. AUTOSAR components and composition, from [12]	90
4.7. Emergence and Detection of Failures vs. Costs, from [54]	91
4.8. AUTOSAR GenericStructure:CommonPatterns:AdminData, from [12]	93
4.9. Example AUTOSAR file structure, from [17]	93
4.10.AUTOSAR Support to enhance the basic conditions	97
4.11.AUTOSAR Data Types Overview, from [12]	98
4.12.AUTOSAR Primitive Data Type, from [12]	99
4.13.AUTOSAR Connectors, from [12]	100

List of Figures

4.14. AUTOSAR Memory Management Overview, from [23]	104
4.15. NVRAM Manager Client/Server Interface, from [24]	105
4.16. NVRAM Manager API function: NvM_WriteBlock, from [24]	106
4.17. Interface interaction of Layers, example “Memory Management”, from [18]	107
4.18. AUTOSAR Implementation of software components and basic software, from [12]	108
4.19. AUTOSAR InternalBehavior of software components, from [12]	109
4.20. Summary meta model excerpt related to modes, from [12]	110
4.21. Kinds of RTEEvents, from [12]	111
4.22. AUTOSAR Resource Consumption: Overview, from [12]	112
4.23. NVM Sequence Diagram for WriteBlock operation, from [24]	113
4.24. NVM Behavior and Runnable Definition, from [24]	114
4.25. EAST-ADL2 System Model: Overview, from [9]	117
4.26. EAST-ADL2 Requirements Modeling: Overview, from [9]	118
4.27. Meta-model for the functional definitions of EAST-ADL2, from [9]	119
4.28. Ports and connectors in the EAST-ADL2, from [9]	120
4.29. EAST-ADL2.0 Support, from [8]	124
4.30. EAST-ADL2.0 Relationship Modeling, from [8]	126
4.31. EAST-ADL2 Support to enhance the basic conditions	129
4.32. EAST-ADL2.0 Behavior Constructs, from [8]	132
4.33. EAST-ADL2.0 Timing Requirements, from [9]	133
4.34. EAST-ADL2.0 Data Types, from [9]	133
4.35. UnSCom Component Description: Overview	140
4.36. Semantical Problem (schematic)	144
4.37. New Concepts: Overview	147
4.38. New Concepts: InformationObject	149
4.39. New Concepts: Function	151
4.40. New Concepts: Process	152
4.41. Ordered Sequence	153
4.42. Parallel Execution	153
4.43. Inclusive Branch	154
4.44. Exclusive Branch	154
4.45. Transformation between multiple representation format, following [50]	155
D.1. Enhanced Basic Conditions by AUTOSAR and EAST-ADL2	182

List of Tables

- 2.1. Comparison of EAST-ADL2 and SAE-AADL, following [37] 30
- 4.1. Comparison between models, textual documents, and code 78
- 4.2. Inconsistencies avoided by AUTOSARs' static semantics 103
- 4.3. Inconsistencies avoided by AUTOSARs' dynamic semantics 115
- 4.4. Inconsistencies avoided by EAST-ADL2s' dynamic semantics 135
- 4.5. Benefits from EAST-ADL2 and AUTOSAR 139
- D.1. Inconsistencies avoided by EAST-ADL2s' and AUTOSAR static and dynamic semantics 183

Chapter 1.

Introduction

1.1. Motivation



Figure 1.1.: Interconnected electronic control units in a modern vehicle [5]

In today's vehicles multiple programmable Electronic Control Units (ECU) are used in order to realize functional requirements of automotive manufacturers or Original Equipment Manufacturer (OEM). A vehicle of the luxury class contains more than 80 of such ECUs, which are networked with each other, as depicted in figure 1.1. Software development for these ECUs is being distributed effectively across several enterprises and countries.

During the integration of developed software with hardware, inconsistencies do emerge constantly -

the software does not work at a first attempt. Cost-intensive fault diagnostics and reworking result out of it. As way out, increasingly methods are being developed, which anticipate integration “virtually”, i.e. on model level.

Virtual integration represents anticipation of integration task on the left side, i.e. the development side, of the V-Cycle development process. Thereby integration should already be realized, for example, during the design phase on the model level. Thus interface compatibilities as well as a later performance within the embedded system can be verified at an early stage of development. For this reason, this thesis analyzes newer modeling and simulation approaches in regard to their application for automotive embedded systems in order to enable virtual integration and to simplify real integration.

1.2. Problems and Challenges

In order to avoid the risk of problems during the integration, the automotive industry has been developing an industry standard called AUTOSAR since 2003. Presently over 100 enterprises are engaged in AUTOSAR, whose integral part is represented by a meta model, which defines an exchange format for software architecture models between OEMs and their suppliers. However, as AUTOSAR only focuses on the technical level, a second project called ATESSST has developed an additional specification, which is able to extend AUTOSAR with additional levels and requirements.

In spite of these efforts concerning the development of standards and specifications, so-called inconsistencies between system components can still hamper integration. Due to the wide-spread landscape of various OEMs and even more suppliers and because AUTOSAR is not a generally applied standard yet, many problems concerning the collaboration of different parties and the integration of system components themselves are complicating integration into an entire vehicle these days. While basic conditions hamper the collaboration and integration of different parties generally, inconsistencies between system components on one single ECU or between several ECUs avoid either static or dynamic networking and make integration impossible.

Therefore the thesis must identify concrete current problems of development, which influence integration, before analyzing AUTOSAR and EAST-ADL2, how far they are able to avoid individual inconsistencies and support the collaboration of different parties. This firstly means to find out problems which come along with:

- a missing general applied standard
- differing tools
- multiple development processes
- a development, which is distributed across multiple countries, companies and developers

And secondly it means to find out, what inconsistencies are avoiding:

- the static interconnection of system components
- the dynamic interaction of system components at runtime

Not until problems will be identified, the thesis has to analyze AUTOSAR and EAST-ADL2 to look for advantages and enhancements, but also for disadvantages and remaining problems. To solve such remaining problems, the meta models may be extended or a mechanism will be identified, in order to reach avoidance of inconsistencies as far as it is possible. Therefore the thesis contributes to recognize or to avoid inconsistencies, while sketching some possibilities for enhancing the collaboration of various parties, which are involved in automotive development, further more.

1.3. Outline

To reach the mentioned goals, this thesis is structured as follows:

First of all chapter 2 introduces AUTOSAR in its section 2.1 and EAST-ADL2 in 2.2. Afterwards section 2.3 details integration in order to get a closer insight into virtual integration, before section 2.4 describes problems and inconsistencies, which hamper integration in general. Finally in this chapter, section 2.5 shows some approaches, which are related to AUTOSAR and EAST-ADL2 in order to simplify integration alternatively.

After that brief overview section 3 describes a survey, which was hold to concretize the abstract problems of the current automotive development, which at this point, have been identified before. The survey's results are formatted within section 3.2, in order to provide a basis for the further course of this thesis.

The problems which will already be identified then have to be analyzed and solutions for them will be shown within chapter 4.1. Therefore section 4.1.1 will show benefits of model based development in comparison with conventional development styles in order to warrant the usage of AUTOSAR and EAST-ADL2 generally. The following sections from section 4.1.2 to section 4.1.5 describe how far AUTOSAR and EAST-ADL2 are able to avoid identified problems concerning the collaboration of parties and concrete inconsistencies between system components. After describing all possibilities of AUTOSAR and EAST-ADL2, section 4.2 carries one necessary point, which is neglected by them at present, to sketch additional language elements for detailing the semantics of AUTOSAR and EAST-ADL2 models.

Chapter 2.

Background and Basics

To support the development and integration of automotive systems, chapter 2 introduces some innovative concepts. Section 2.1 introduces AUTOSAR dealing with development and the specification of automotive system architectures, closely related with implementation level. To complement AUTOSAR with missing or additional artifacts and more information beside this technical level, section 2.2 presents the concept of an Architecture Description Language (ADL), while introducing a special ADL called EAST-ADL2. AUTOSAR as well as EAST-ADL2 are model-based solutions, whereby models represent components building the system. One of the goals of AUTOSAR and EAST-ADL2 is to support the integration of models similar to the integration of implemented components. Before section 2.3 puts the notion of integration in concrete form. Afterwards, some general problems hampering the integration of components on model or implementation level are discussed. While the problems are detailed in section 2.4, the later chapters of this thesis will analyze how far AUTOSAR and EAST-ADL2 are able to avoid them on design or model level already. Finally, section 2.5 introduces some existing and related concepts enabling integration.

2.1. AUTOSAR

Automotive software development is characterized by a widespread landscape of different OEMs and suppliers. A global distributed development of components as well as a missing standard for development leads to individual solutions of different suppliers. But individually created solutions are mostly not able to interact with other components or they are not able to be integrated into the entire automotive system in a first attempt. Furthermore, the exchange or the upgrading of components are difficult.

Having these problems in mind, an initiative called AUTOSAR was founded in 2003. AUTOSAR stands for Automotive Open System Architecture and establishes a standard modular software infrastructure for application and basic software. This enables exchanging parts of the system's software and describing an embedded automotive system on a technical level close to implementation [52]. The standard allows to describe all properties of the software and the hardware of an embedded automotive systems by a common format. AUTOSAR enables modularity, scalability, transferability and re-usability of software among projects, variants, suppliers, customers, etc. Generally, the main objectives of AUTOSAR are:

- To manage increasing E/E (electric/electronic) complexity associated with growth in functional scope
- To improve flexibility for product modification, upgrade and update
- To improve scalability of solutions within and across product lines
- To improve quality and reliability of E/E systems
- To enable detection of errors in early design phases

To reach these objectives and in order to control complexity at the same time, AUTOSAR defines several self-contained description documents. These documents precisely describe certain parts of the architecture and affect all relevant properties of an entire component based system. Dependencies between these descriptions as well as a form of guide line for building these documents are described by the AUTOSAR Methodology [13]. The descriptions themselves, i.e. how they have to be specified, are defined by AUTOSAR by means of a meta model [12]. The AUTOSAR Architecture, the Methodology, and the Meta Model are discussed more closely below.

2.1.1. AUTOSAR Architecture

Figure 2.1 depicts the software architecture specified by AUTOSAR. It is a layered and component based architecture as well. Above the concrete hardware of microcontrollers or ECUs AUTOSAR specifies three Abstraction Layers: Basic Software Layer, AUTOSAR Runtime Environment and Application Layer.

The Basic Software Layer, which is situated below the AUTOSAR Runtime Environment, abstracts the hardware and provides services to AUTOSAR Software Components by standardized and ECU specific software components. In order to enable exchangeability of services and for customization

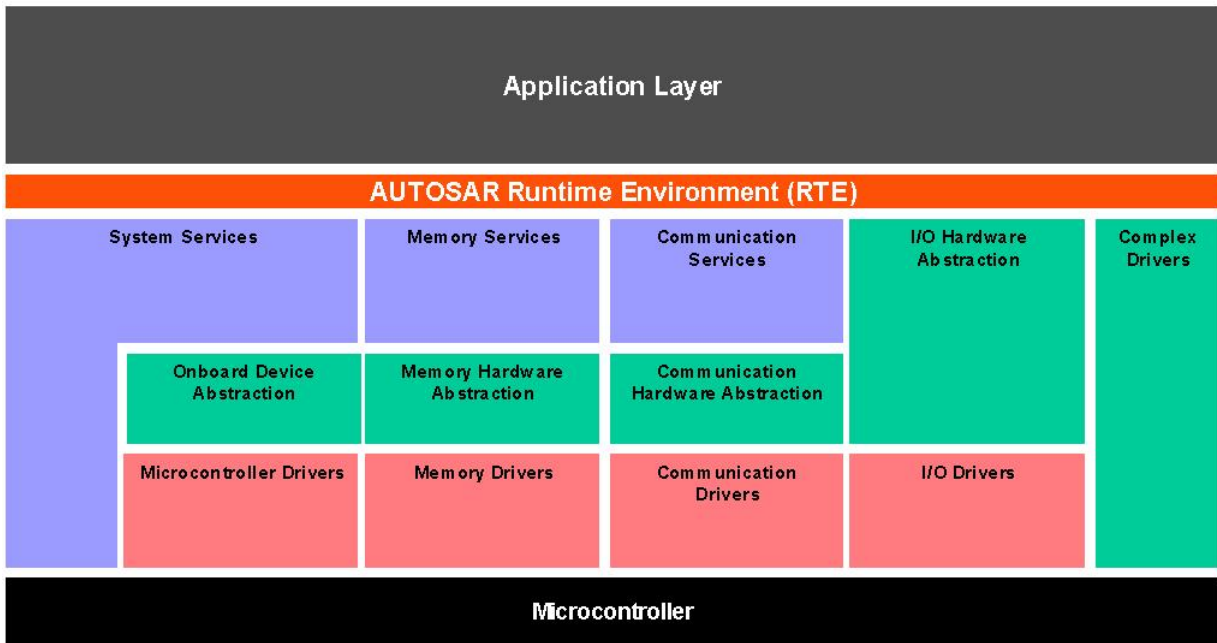


Figure 2.1.: AUTOSAR refined layered software architecture from [16]

of different supplier specific ECUs, the Basic Software Layer is further detailed into three abstraction layers (Service layer, ECU Abstraction Layer, Microcontroller Abstraction Layer), plus the possibility to implement Complex Device Driver which cannot be mapped into a single layer. Figure 2.1 shows that there are different abstraction layer stacks, whereas each stack stands for an individual function, like memory, communication, I/O or device driver functionality.

The next level of AUTOSAR architecture is the RTE layer. This layer is realized by two formings: As Runtime Environment(RTE) and as Virtual Functional Bus(VFB). The RTE realizes the VFB on a concrete ECU and is explicitly generated for each ECU, like depicted in figure 2.2, which shows the relation between VFB and RTE aligned with the basic AUTOSAR approach.

In order to fulfill the goal of relocatability, AUTOSAR Software Components are implemented independently from the underlying hardware. This independence is achieved by providing the VFB. The VFB provides a virtual hardware, mapping independent system integration, and abstraction of the AUTOSAR Software Components interconnections. Hence, communication between different software components and between software components and their environment (e.g. hardware driver, operating system, services of the basic software layer, etc.) can be specified independently of any underlying hardware (e.g. BUS systems or microcontrollers). This is also shown in Figure 2.2.

Thereby communication is possible through the concept of standardized AUTOSAR interfaces, which

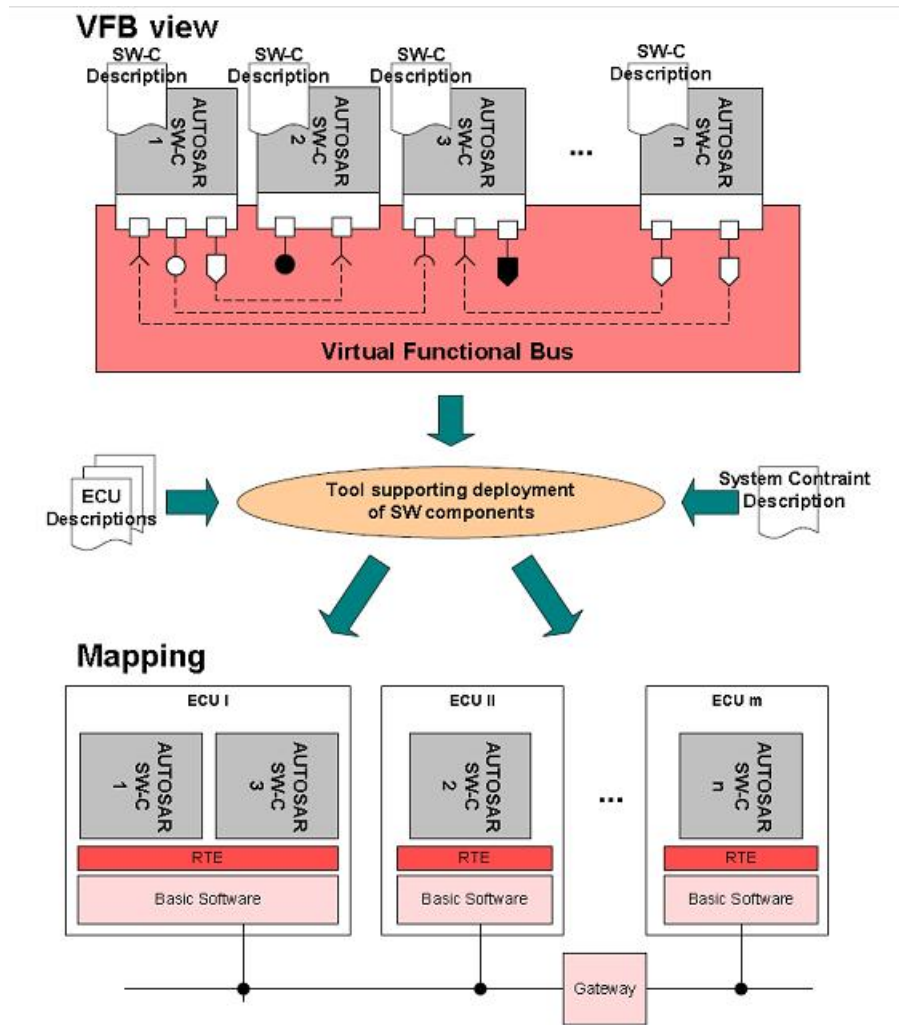


Figure 2.2.: Basic Autosar Approach from[16]

encapsulate each component. The interfaces enable the integration of AUTOSAR Software Components in that way that parts of the integration process of automotive software can be done at much earlier design phases compared to today's development processes [16].

Upside the RTE the Application Layer contains application software components. Application software components use the services of the Basic Software Layer and communicate with other components over the RTE or VFB exclusively.

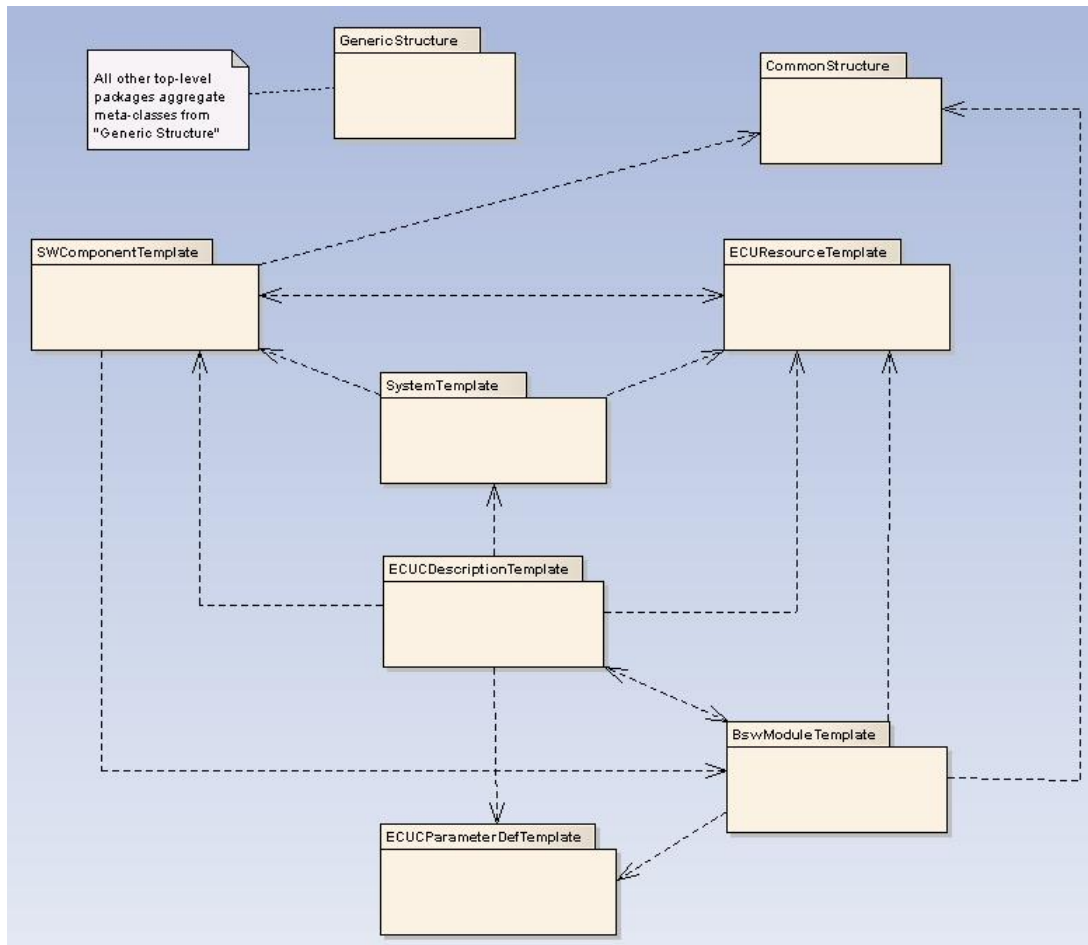


Figure 2.3.: AUTOSAR Templates: Overview from [10]

2.1.2. Meta Model and Templates

In order to represent the information describing the aforementioned architecture and its components via a standardized and machine readable format, AUTOSAR defines a meta model [12] in terms of a Model Driven Engineering (MDE). MDE is the systematic use of models as primary engineering artifacts throughout the engineering lifecycle. This enables advantages of a model-driven approach like model transformation and automated code generation in the automotive software development. Furthermore, this approach specifies a Domain Specific Language (DSL) which satisfies special requirements of an automotive system. For this purpose, AUTOSAR defines an abstract syntax by the means of its meta model which defines all describable AUTOSAR concepts necessary for specification of such automotive systems. This meta model is described using the UML, whereas a special UML Profile ([10]) was created, which can be applied to almost any UML tool in order to use AUTOSAR syntax.

Furthermore the meta model and its language elements is subdivided into packages like depicted in figure 2.3. These packages are called AUTOSAR Templates and basically there are three main description templates to describe information concerning software components, ECU resources, or the entire system. These Templates are detailed in the following.

AUTOSAR Templates

The AUTOSAR Templates specify the syntax to describe parts of the AUTOSAR architecture. Figure 2.3 depicts an overview of the Templates and relationships among them. The template, which holds all parts together is the System Template [25]. As the name implies, it describes properties of the entire system. Generally, a filled template defines the relationship between the pure Software View on the System and a Physical System Architecture with networked Electrical Control Units (ECU) instances. By means of the system template, five major elements can be defined: Topology, Software, Communication, Mapping and Mapping Constraints. The Topology part of the system template describes the physical System Topology of a vehicle modeled in AUTOSAR. This is formed by a number of so called ECUInstances which are interconnected to each other in order to form ensembles of ECUs.

Furthermore, the system template contains a software composition element containing all application software which are specified to run on a particular ECU. Each of these application software components is defined more closely inside of the Software Component Template [15]. In order to distribute the software over ECUs, the system template defines a so called SystemMapping which maps application software components to certain ECUs. Beyond the software to ECU mappings the SystemMapping also maps the data exchange between software component signals as well as the way a signal should take between software components. However, the SystemMapping also contains further relevant mappings and elements to describe the communication using signals, frames and PDUs which are explained more closely in the SystemTemplate Specification of AUTOSAR.

A further important template to describe an AUTOSAR system is the ECURessourceTemplate [19]. It provides the syntax for describing and checking the consistency of characteristics and features of automotive ECUs. This template is used to specify the hardware of ECUs in detail. Hardware Ports, Memory, Processing Unit, Peripherals and other Electronics of an ECU are described by the means

of the ECU Ressource Template which also depends on the SystemTemplate.

All these templates together build the core of the meta model, but there are much more templates describing information of an AUTOSAR system in more detail. They are called GenericStructure , CommonStructure, ECUDescriptionTemplate [11], BswModuleTemplate and ECUCParameterDefTemplate. The dependencies among these documents or templates are prescribed by the AUTOSAR Methodology [13], which is described in the following.

2.1.3. Methodology

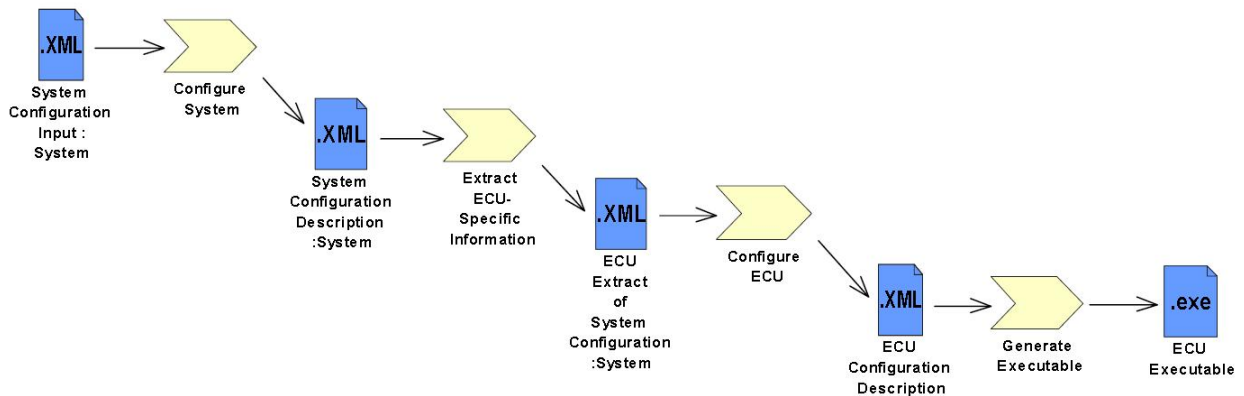


Figure 2.4.: Autosar Methodology: Overview from [16]

The AUTOSAR methodology [13] does not describe a concrete proceeding or development process. It defines a recommended workflow, which specifies when certain information has to be available for further processing. This workflow describes independently of roles, responsibilities or detailed time specifications, which information depend on each other and which information are a precondition for other documents. In this context AUTOSAR uses the term work-product for such an information object. Additionally, the methodology describes which work-products have to be brought together in order to generate new, more detailed work-products out of several input work-products by specified activities. These steps can partly be automated. The processing of several work-products goes through different activities until, at last, all necessary information, which describe the entire system, are arranged by the mean of multiple descriptions. This means that the methodology describes dependencies between different information for building an entire system and steps for generating new work-products inside a workflow. I.e. when information (some work products) are completely available as input, new output can be generated. This workflow runs through the whole developments phase, from system design to Executable ECU Code.

Figure 2.4 depicts a simplified view of the AUTOSAR methodology. The blue documents labeled with XML represent work-products. These work-products are specified by means of the AUTOSAR Template components. Using a schema generator, a XML interchange format can automatically be generated from the meta model or rather from the model itself. For this reason, AUTOSAR specifies so-called Model persistence Rules for XML [14] so that all models can be exchanged by the established standard XML. Further work products may be object code, header files or others. On the other side, the arrows between work-products define process steps for transforming or gathering of necessary information. Details on the AUTOSAR methodology, i.e. further process steps, work-products, and dependencies, may be found in the AUTOSAR Methodology specification [13].

2.2. Architecture Description Languages

AUTOSAR is a detailed language to describe embedded systems. However, as mentioned above, AUTOSAR explicitly focuses on a technical description close to implementation of such systems. This focus disables AUTOSAR to support an overall development process well and to share information with other stakeholders. It is not possible to contemplate a system from another view than the technical. However, such views may support development stages such as analysis or design, and they describe, for example, common or non-functional requirements. Furthermore, AUTOSAR insufficiently addresses behavior or error modeling, validation and verification of models. To extend AUTOSAR with additional or necessary abstraction levels or description elements, it is possible to factor an ADL into AUTOSAR. Generally, the availability of an enhanced and standardized ADL and a methodology for using it will facilitate communication, analysis and synthesis of automotive embedded systems.

In the following the concept of an ADL and the specific ADL EAST-ADL2 is introduced.

2.2.1. ADL Concept

ADL stands for Architecture Description Language and, in general, it is a language to describe software architectures. We define software architectures as follows:

“Software architecture is a level of design that involves the description of elements from

which systems are built, interactions among those elements, patterns that guide their composition, and constraints on these patterns.” [45]

So an ADL has to detail software architectures concerning (sub-) systems and interconnections among them. On the one hand it has to describe the system structure. On the other hand it deals with components's behavior and their interaction inside of a system. However, an ADL gathers not only information for documentation but also for tool supported further processing of such information (e.g. model analyzes, simulations, code generations, model transformations,...)

There are a lot of approaches called ADL, like Aesop [33], ArTek [57], C2 [44], Darwin [42], LILEANNA [58], MetaH [60], Rapide [41], SADL [46], UniCon [56], Weaves [35], and Wright [6], SysML [4], EAST-ADL [7], and SAE AADL [3]. However, there is no clear standardized definition. Therefore, the aforementioned ADLs differ on several points, which are discussed below.

Artifacts of an Architecture Description Language

First of all, ADLs differ in what they model and how detailed these models are. There is an agreement that an ADL describes **components**, **connectors**, and the **configuration (Topology)** of a system. Furthermore, to be usable and applicable, it must provide tool support for architecture-based development and system evolution. Some ADLs focus on a special domain, detail individual components more than others, or provide a more formal language than others. (E.g. While Rapide focuses on the external behavior of interfaces, Wright details semantic of architectural connections). Another questions which has to be answered by an ADL is: How to support developers?

An ADL should enable communication between developers and does support understanding for the entire system. In terms of this, an ADL has to be simple, understandable and should have graphical syntax. Moreover, it should provide the kinds of tools that aid visualization, understanding, and simple analyses of architectural descriptions.

In certain cases it is also necessary to describe a system from different point of views. This enables different stakeholders participating in the development, to get specialized insights into system details which are explicitly relevant for their work. I.e. on the one hand an ADL has to describe the technical architecture, which must be communicated to software developers. On the other it should

also be possible to describe a functional architecture or non functional requirements. This kind of architecture is communicated to stakeholders and enterprise engineers. To meet the requirements of any stakeholder, an ADL can provide possibilities to specify the entire system on multiple levels of abstraction or views. These levels especially address requirements and expertises of certain stakeholders and contain information for their work. Concentrated information concerning different aspects or levels of a system description are characterized as artifacts. An artifact is the result of a specific analysis, i.e. a particular view. Thus it must be clarified for each artifact, **what** it describes from which **point of view** and **how** it supports the developer. By this means, a system is composed of multiple artifacts step by step.

Language Elements of an Architecture Description Language

Unlike textual descriptions, an ADL must have a formal syntax and semantic, to enable analysis, model checking, parsing, compiling, and code generating. However, some ADLs exclusively use strict formal methods, like Petri Nets or Process Algebra, to achieve the most formal semantics as possible for describing an architecture. This strict formal proceeding comes along with the advantage that the descriptions of a system can preeminently be analyzed and simulated. However, on the other side, these approaches are very difficult to learn/apply and thus they are not widely accepted. By contrast other ADLs use semi-formal approaches to ease understanding and application of such a language. Semi-formal approaches base for example on UML and describe an abstract syntax of architecture elements. By means of constraints and limiting profiles these approaches can be extended with additional semantics. Hence, such approaches are also able to provide adequate formalisms as well as tool support.

Some ADLs also describe a methodology to provide developers with a form of Guide Line. Such a methodology describes how and when certain artifacts have to be created and so it specifies a proceeding comparable with a development process. Therefore the methodology shows correlations between artifacts inside the proceeding as well as when artifacts have to be created. It answers the questions how to build an entire system using the ADL.

Thus an ADL can generally be defined as:

A language that provides features[, like language elements, views, artifacts, and/ or methodologies,]for modeling [and analyzing] a software system's conceptual architecture, distinguished from the system's implementation. ADLs provide both a concrete syntax and a conceptual framework for characterizing architectures. The conceptual framework typically reflects characteristics of the domain for which the ADL is intended and/or the architectural

style. The framework typically subsumes the ADL's underlying semantic theory (e.g., CSP, Petri nets, finite state machines).(compare Mevidovic [45])

2.2.2. EAST-ADL2

This section wants to introduce a specific ADL developed for the automotive domain. This ADL, called EAST-ADL2, has been developed by multiple agents in the automotive domain for several years, including both OEMs and suppliers.

In a previous version EAST-ADL2 was developed within the scope of the EAST-EEA ITEA project. The developed architecture should ensure interoperability between the software and the hardware to enable re-usage and exchangeability of distributed components. In order to simplify development of embedded automotive systems, a subproject of EAST-EEA developed a domain specific ADL called EAST-ADL. The ITEA project finished 2004 and was further developed within the ATESSST, Advancing Traffic Efficiency and Safety through Software Technology, IST-project. The current version of EAST-ADL is version 2.0. Now a new project called ATESSST2 follows.

EAST-ADL2 Abstraction Levels and System Model

EAST-ADL2 is an UML based solution and an UML profile captures the ADL in a XML file. This enables using well-known tools for tool interaction and model exchange as well as open use and standardization. Furthermore, it also complements AUTOSAR with e.g. functional specification and requirements. Beyond the objective of EAST-ADL2 to define a standardized Architecture Description Language for modeling all aspects of an (automotive) system, it also targets documentation and a methodology.

EAST-ADL2 Language Elements EAST-ADL2 defines language elements in order to describe certain properties of an architecture. These language elements can be classified into five main concerns:

- **Requirements:** The Elements of the requirements language base on SysML constructs and are used to specify all kind of requirements. A requirement is a condition or a capability which must be satisfied by the system. Requirements change over time and they could be introduced by various people, like marketing people, control engineers, system engineers, software engineers, Driver/OS developers, basic software developers or hardware engineers.

- Structure:** This part of the specification defines the structural constructs used in EAST-ADL2. The structural view of a model focuses on the static structure of components of the system being modeled and their static relationships. This includes the internal structure of such components like their external interfaces through which they can be connected to communicate with each other, by exchanging data or sending messages. For the design of systems of arbitrary size and complexity, the possibility of hierarchical structuring of the instances is provided in the language. The structure contains the most important packages and bases on the BasicComponents package of UML.
- Behavior:** The language elements here are used to describe a behavior model. EAST-ADL2 supports its own simple algorithmic behavior model and it is also able to reference external defined behaviors. That makes it possible to reference behavior definitions, which may be specified by other tools and/ or mechanisms than EAST-ADL. (E.g. Statemate or Simulink) Beside behavior modeling the EAST-ADL2 enables Error-Modeling as well.
- Verification and Validation:** These elements concern possibilities to describe Testing and Verification strategies or methods.
- Variability:** Elements of the variability package are used to support describing various variants of architecture's artifacts or product lines.

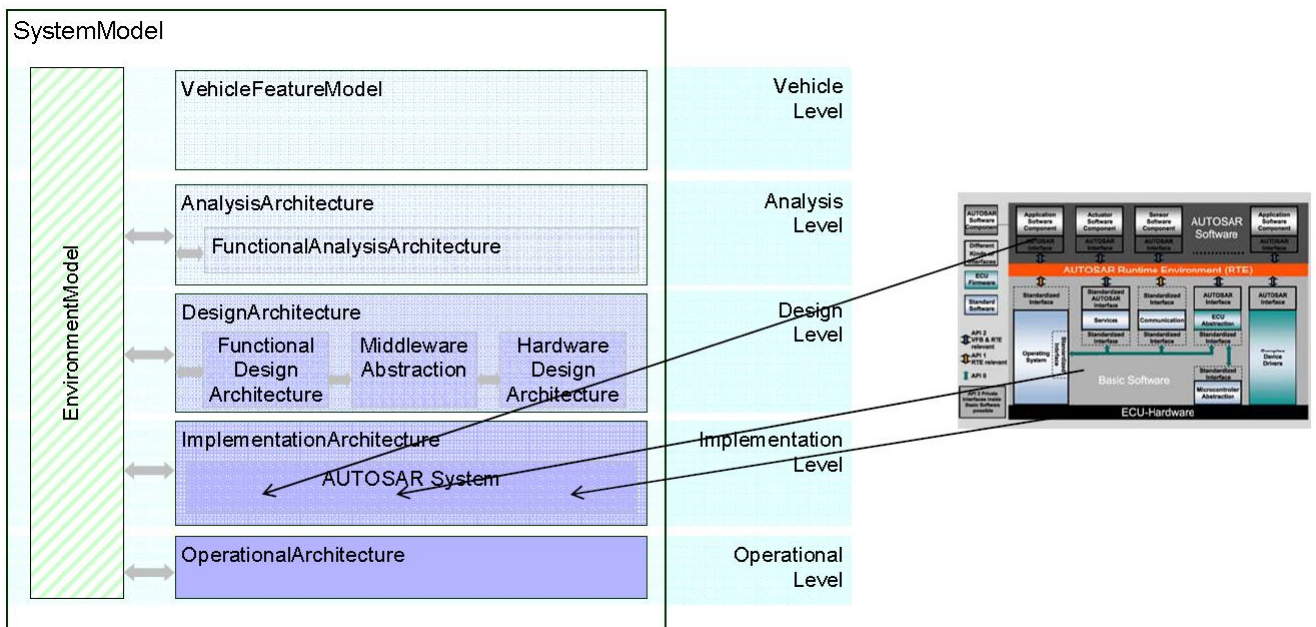


Figure 2.5.: EAST-ADL2 Abstraction Layers and Artifacts from [9]

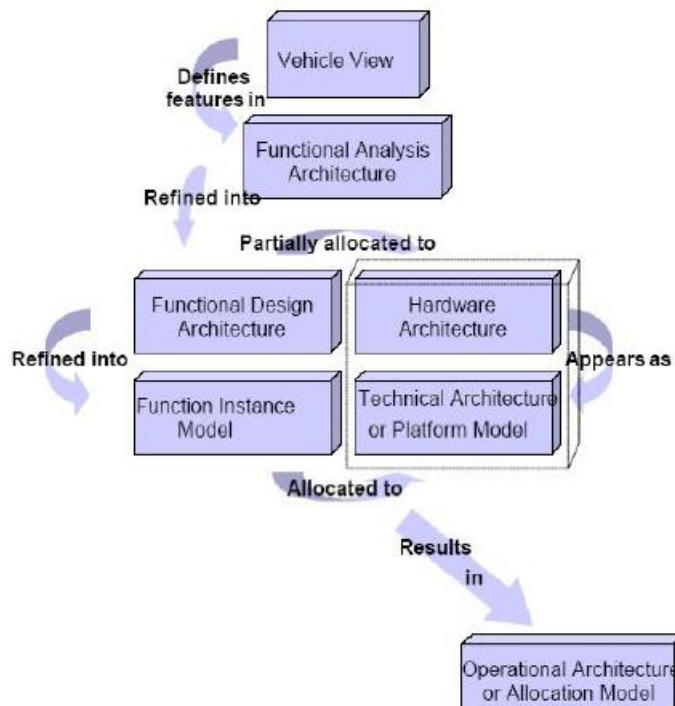


Figure 2.6.: The seven artifacts of EAST-ADL2 and their interactions: Overview from [37]

EAST-ADL2 Artifacts and Abstraction Layers EAST-ADL2 abstraction layers are used to allow reasoning of the features on several levels of abstraction. The aforementioned language elements are used by several abstraction layers, and so they can be refined from a general view down to a more specialized view. Figure 2.5 shows the five abstraction levels of EAST-ADL2: Vehicle Level, Analysis Level, Design Level, Implementation Level, and Operational Level.

These abstraction levels are populated by 7 artifacts, i.e. the process elements, which are described by the language. The levels and artifacts have been selected to allow various, company specific development processes and enhance the separation of software and hardware development.

Figure 2.5 also shows that there is a relationship between AUTOSAR and EAST-ADL2 concepts. In particular, the figure shows that AUTOSAR especially concerns the implementation level of EAST-ADL2.

Firstly there are five artifacts building multiple abstraction levels which abstract technical details of

an architecture (or rather of AUTOSAR):

- **Vehicle View:** The Vehicle Level is the most abstract Level inside of EAST-ADL2.0. Here the VehicleFeatureModel(VFM) supports model-based requirements engineering. The electrical systems, vehicle product lines as well as perceivable features of a vehicle are described by use cases. So, general needs and requirements are defined to be part of the VFM.
- **Functional Analysis Architecture (FAA):** One or several entities (analysis functions) of the FAA can be combined and reused to realize vehicle features of the VFM. The FAA captures the principal interfaces and behavior of the subsystems of the vehicle. It allows validation and verification of the integrated system or its subsystems on a high level of abstraction.
- **Functional Design Architecture (FDA):** The aspects, which orient more towards implementation are introduced while defining the Functional Design Architecture (FDA). The features are here realized in a more implementation-oriented manner. The system is structured and decomposed to suitable components taking efficiency, legacy and reuse, COTS availability, hardware allocation, etc, into account.
- **Middleware Abstraction (MWA) or Platform Model:** The Middleware Abstraction artifact contains software components realizing middleware und OS-Functionality of AUTOSAR. Since AUTOSAR has based on a standardized platform with fixed content, the entities of the middleware does not have to be modeled explicitly in the AUTOSAR context. AUTOSAR middleware modules are predefined and can partly be generated based on requirements of application software.
- **Hardware Design Architecture:** The Hardware Design Architecture artifact describes the topology of an architecture. This concerns ECUs, sensors/ actuators, BUS systems and gateways. The respective AUTOSAR entities are found in the ECU Resource Template and System Template (see section [2.1.2](#)).

Below these levels there are two further artifacts to refine the artifacts from above, in order to describe the technical level of implementation:

- **Implementation Architecture (IA) or Function Instance Model (FIM):** The Implementation Architecture artifact contains software components realizing the functionality of application software specified by the FDA. The content of the IA is defined by the means of the AUTOSAR Software Component Template (see section [2.1.2](#)).

- **Operational Architecture or Allocation Model:** Operational level representing the binary entities and their related tools. It represents the actual software and electronics in the manufactured vehicle.

Cross-architecture constructs, such as allocation decisions and signal-to-frame mappings have to be part of the Implementation Architecture directly, as both software and hardware should be allocation independently and reusable. Parts of the AUTOSAR System Template (compare to section 2.1.2) of AUTOSAR are therefore located directly in the Implementation Architecture.

To verify and validate a feature across all abstraction levels, an **Environment Model** is needed early on. This plant model captures the behavior of the vehicle dynamics, driver, etc. on each abstraction level.

Figure 2.6 exemplifies the interplay between different artifacts of EAST-ADL2. Thereby some kind of methodology is introduced in order to prescribe an chronological order between artifacts and activities. This methodology is described in more detail below.

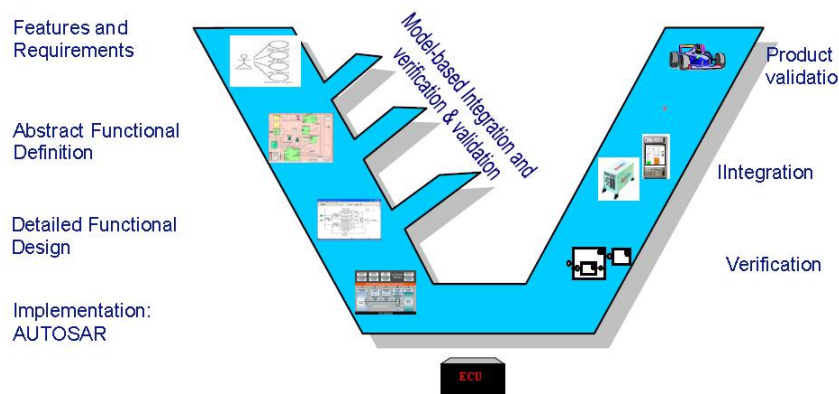


Figure 2.7.: A V-model for automotive development from [7]

EAST-ADL2 Method aligned with the V-Model Figure 2.7 shows that EAST-ADL2 may be aligned with a common V-Model. As such, it provides a form of methodology for creating the different artifacts. First of all, the VehicleFeatureModel matches the Features and Requirements Level of the shown V-Model and models the intentions, which should be satisfied by the system. The Functional Analysis Architecture can be compared with the next level down the left side of the V-Model, the

abstract functional definition. This uses the results from the level above and models the common system specification. Based on these abstract functional definitions, the Functional Design Architecture of EAST-ADL can be used to model a system architecture and a detailed functional design as well. These levels capture information before module specification and implementation. After that, the more specific artifacts for implementation specifications of EAST-ADL can be derived from the former specifications. For this purpose, EAST-ADL2 specific “ADLRealization”-dependencies between abstraction levels are used to record traceability from abstract levels down to more specific levels.

2.2.3. EAST-ADL2 vs. SEA-AADL and other approaches

Section 2.2.1 lists a lot of ADLs. As such, each of them enable a description of common architectures containing components and interconnections among them. But the nature of these ADLs is mostly either aligned with a theoretical and academical background or not aligned with specific domains or both in the majority of cases. For this reasons and because there are two languages already, EAST-ADL2 and SAE-AADL, more specific to the needs of automotive system development, just these two approaches will be discussed below.

The SAE-AADL language comes closest to the facilities of EAST-ADL2(compare section 2.2.2). IN 2004 SAE-AADL was renamed from “Avionic Analysis and Design Architecture” to “Architecture Analysis and Design Language”. Although it was developed by SAE, the Society of Automotive Engineers, it was originally intended for avionic systems. However, by now the AADL is also being used within other domains, such as Space, Robotics, Industrial Control, Medical, and Automotive. Table 2.1 bases on [37] and clarifies big differences and commonalities of these two languages.

SAE-AADL is a more common approach usable for various domains than EAST-ADL2. However, a comparison between the two languages shows that EAST-ADL2 can complement SAE-AADL, and vice versa. Both languages enable modeling of static, software-intensive embedded systems. On the one hand, while SAE-AADL can be used for modeling and analysis on a level close to implementation, it is possible to use EAST-ADL2 for design modeling and validation on a high level of abstraction. Especially, well supported possibilities for analysis and tool usage make SAE-AADL interesting. EAST-ADL2 though, provides theoretical concepts for analysis, but there are just a few tools supporting or executing these analyzes. On the other hand EAST-ADL2 supports modeling of behavior, which is not considered by SAE-AADL. This makes it possible to integrate behavior models, like State-Charts- or Use-Case-Diagrams into a EAST-ADL2 model. Additionally, EAST-ADL2

	EAST ADL2	SAE AADL
Focused Domain	Automotive	Main focus on Avionic systems. But also used by other domains, like Automotive, Robotics, Space flight,..
Focus on distributed embedded systems	yes	yes
Focus on dynamic systems	no	yes
Modeling of non architectural elements	requirements	not supported
Modeling of Behavior	supported by External or Native Behavior	not well-supported
Support of HW-/SW-Co-Design	yes	yes
Notation	graphical, a few textual	graphical, textual
Abstraction Levels	multiple abstraction levels	no specified abstraction levels
Interfaces to other languages	XML, UML Profile, External Behavior	XML, UML Profile
Exchange of models	XML/ XMI	XML/XMI
Development process	V-Model is base	no specified process
Support for analysis	possible through Validation and Verification language elements, see 2.2.2 . But not mature or well-supported by tools	yes, e.g. performance, schedulability, reliability..

Table 2.1.: Comparison of EAST-ADL2 and SAE-AADL, following [37]

provides a variant management to support product lines and requirements modeling.

2.3. Integration

A lot of hardware, software, and mechanical components must be integrated in order to realize all functions of a vehicle. EAST-ADL2 and AUTOSAR want to enable an integration of such components on the model level already without using expensive real components on an early stage of development. However, for understanding model integration firstly, it is important to understand integration itself. For this purpose chapter 2.3.1 introduces integration and some questions, which must be observed for integration, generally. Afterwards, section 2.3.2 details the term integration to show differences between integration within an automotive environment in comparison to more general component based integration strategies. After that, section 2.3.3 introduces the concepts of view integration as well as model integration, as these kinds of integration are of particular interest to the integration on model level of AUTOSAR and EAST-ADL2. Finally section 2.3.4 shows some practical integration problems in the scope of AUTOSAR.

2.3.1. Integration in general

Generally integration is derived from Latin "integrare" and means to bring different artifacts together into a whole. In other words: it is a process of combining or accumulating. Because of this broad definition, integration is a term which is used within many scientific areas whereas each of it has its own specific interpretation for it. However, two basic questions concerning integration remain the same independently from individual areas:

- What has to be integrated?
- How does something have to be integrated?

As the thesis' goal is to address component based software integration, the following sections also will discuss these questions to that effect.

What has to be integrated? In general, from a software point of view two disciplines of integration concerning the "what" can be identified.

- On the one hand, system integration is a process by which smaller pieces of software are brought together to form a larger piece of software. In this context, a piece of software means a software component, whose more detailed definition can be found, for example, in [39] or [43]. This kind of integration deals with the interconnection of components' functionality. In today's software engineering this generally means the interconnection of ports and interfaces of several components that they are able to communicate or to interact over messages and signals.
- On the other hand data have to be integrated as well. This kind of data integration allows data from one device or software to be read or manipulated by another. Especially in the area of databases this kind of integration is established well. The problem here is to match different data representation formats as well as exchange formats and semantics of data.

How does something have to be integrated? The kind of integration also has a strong stake in efficiency of integration because it is a big difference if data or functions have to be integrated when an executable code is available or before. This leads to some integration strategies, which will be discussed in more detail below:

- Manual integration of data and functionality is the hardest way one can choose because of the complexity of large systems. Therefore this is not really a possibility and so it is not further discussed by this thesis.
- Other approaches, such as [40] and [28], describe a good taxonomy of common integration strategies, which can be applied on code/software level directly. This kind of integration can be seen as logical integration, which applies to integration during execution, where the data remain on the different sources and integration only occurs when data or functions are requested at runtime.

Each term of this taxonomy stands for a specific class of design patterns, which are able to manage the integration of components during execution. Such a design pattern can either be a bridge, which converts data from one component to another, without knowing its clients, or a wrapper, which is structured in that way, that it is completely shielding each component from direct interaction with its context. Further design patterns are, for example, the proxy or the mediator pattern.

- Beside these essential design patterns there are other possibilities to overcome integration before a code is available. These approaches, which can be seen as materialized integration, stand for integration before execution and thus during the design or analysis phase. Such an approach tries to eliminate later integration faults by the use of more detailed descriptions. In comparison with logical integration, all data and functions, which have to be brought together, are prepared before an entire system executes the functions or works with their data. These approaches detail properties of a system to that extend, that misunderstandings between developers concerning components or artifacts may be reduced or solved before later conventional integration phases. Especially, by the means of models, these detailed description techniques are realized in order to enable integration already on model level, which is discussed more closely in section 2.3.3. Some examples for common techniques are:

- Specification of extended interface descriptions, compare [50]
- Interaction protocols using for example the Protocol Definition Language (PDL) [50]
- Formal description languages like petri nets, compare [59, 55]
- Semi-formal description languages like UML, AUTOSAR, or EAST-ADL2 concerning static structures or the dynamic behavior of a system (compare: Sequence Diagrams, Activity Diagrams, Use Cases Diagrams, State Machine Diagrams, or Interaction Diagrams)

Informal descriptions like Text may also describe system components, but they are not listed here due to their lack of the ability of computational processing as well as formalism.

2.3.2. Integration in an automotive environment

After the last section has dealt with common questions of a general component based integration, the following shows what integration means for automotive systems. For this reason and in order to get a better understanding for what integration means in the scope of this thesis, the following section will describe step by step what is special for integration in a vehicle. For this purpose, the most important points of integration of common component based applications are described at first. After that, the challenges of an integration of components within embedded systems are discussed, before automotive systems as special case of an embedded system are addressed. These three forms of integration can be seen as specialization from the more general case of component based

applications down to the more special case of component based application in an embedded automotive system. Therefore the respective requirements, which come along with the more general case, must be taken into account by the more special case of integration.

- **Integration in general component based applications:** Today, applications are not developed independently. Most applications are part of larger architectures and have to interact with other applications. While some applications are developed for a specific application fields from scratch, other applications just are reused in form of components of the shelf(COTS). This fact must be taken into account for the development of components, too, since there could be several functional dependencies between individual applications within one architecture. During components' development it has to be ensured already, that the components can be integrated into the entire architecture afterwards and are able to interact with other components as well. This concerns **system integration** and **data integration** like described in section 2.3.1. All these points considerably influence a good and accurate integration of components into an entire architecture.
- **Integration in an embedded environment:** Unlike general component based applications, an embedded system is a computer, which is embedded into a technical context and responsible for controlling, setting, and observing that system. They often are adjusted to a special job and therefore they can be realized as a cost-effective composition of hardware and software. In the process, an embedded system is subject to strong restricting conditions, such as minimal costs and a low consumption of space, energy, and memory. While an embedded system normally has just reduced resources, mostly without any storage drive, operating system, keyboard, or display compared to a personal computer, an embedded system mostly has to comply critical requirements, like real time or safety requirements.
The more functionalities provided by an embedded system ,the more is the complexity of integration of such functionality. In consideration of the restricted resources, like described above, the software has to be attended to them.
- **Integration in an automotive environment:** A vehicle or an aircraft are examples for a complex embedded system and a special case of conventional embedded systems. Such a complex embedded system is a network of autonomous embedded systems in combination with mechanic as instruments. In the concrete case this means, that multiple ECUs are distributed over the whole vehicle, whereas each ECU represents a conventional embedded system itself, which has to interact with other ECUs or rather other embedded systems. For this reason, the ECUs are interconnected by so called BUS systems, like CAN, LIN, or FlexRay. This networking of multiple embedded systems, which controls the hardware, software and mechanics

functionality, represents the particular challenge for integration within the entire system vehicle.

AUTOSAR component integration

AUTOSAR as description language for complex automotive embedded systems has to deal with the following kinds of integration:

- **Software-Software Integration(SW-SW integration):**

Software-Software Integration means that software or software components designed by AUTOSAR have to be integrated into a whole, so that an interaction with other software within the system or vehicle is possible. However, AUTOSAR specifies different kinds of software components. Each kind of software component must be integrated with similar or other kinds of components. The resulting integration strategies are described below:

- **Application Software Integration (ASW integration):**

On application layer, see figure 2.1, there are many application described using the AUTOSAR software component Template, see section 2.1.2. Each AUTOSAR software component specifies its required and provided interface, the behavior, and a lot of other properties, which are important to describe the interplay with other components. During the integration phase, all application software(their descriptions) have to be put together. The difficulty is, that interaction points between components are described correctly, so that an interplay is possible.

- **Basic Software Integration (BSW integration):**

On basic the layer, the software to manage basic functionality, like memory, communication, or scheduling is established. Today's basic software layer contains 53 modules to realize any function which is important to support all functions of application software. Since basic software modules also have to interact, the basic software modules have to be integrated as well.

- **Application Software - Basic Software Integration (ASW-BSW integration):**

The two blocks of the application layer and the basic software layer have to be brought together in that way, that an application software component can also use the infrastructure functionality provided by the basic software. As a kind of middleware between application software and basic software, on RTE layer the runtime environment can be

generated for a specific ECU. I.e. to integrate the different layers of the AUTOSAR architecture into a whole, so that any software component is able to communicate with other components like specified by a developer.

Especially, interfaces are the interaction points between components and therefore have to be interconnected during integration. Therefore AUTOSAR specifies three different types of interfaces, see figure 2.8, to interconnect the components which were just described.

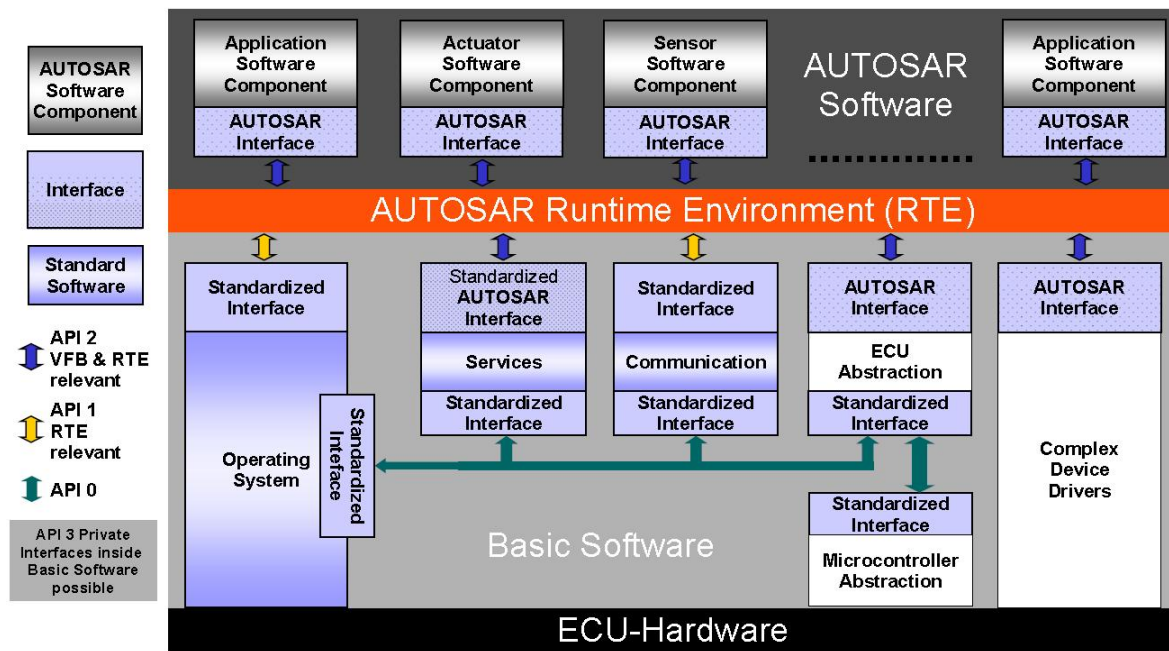


Figure 2.8.: AUTOSAR Interfaces from [16]

– **AUTOSAR interfaces:**

An “AUTOSAR Interface” and its description is independent of a specific programming language, ECU or network technology. AUTOSAR Interfaces are used in defining the ports of software-components. AUTOSAR makes it possible to implement this communication between software-components either locally or via a network.

– **Standardized AUTOSAR interfaces:**

A “Standardized AUTOSAR Interface” is an “AUTOSAR Interface” whose syntax and semantics are standardized in AUTOSAR. The “Standardized AUTOSAR Interfaces” are typically used to define AUTOSAR Services, which are standardized services provided by the AUTOSAR Basic Software to the software-components.

– **Standardized interfaces:**

The “Standardized Interfaces are typically defined for a specific programming language (like “C”). Because of this, “standardized interfaces” are typically used between software-modules which are always on the same ECU. When software modules communicate through a “standardized interface”, it is not possible any more to route the communication between the software-modules through a network.

Beside software integration, there are further elements, which must be integrated. Although they are not of any particular interest for this thesis, they are listed for completion in the following:

- **Software-Hardware Integration (SW-HW integration):**

As a vehicle can not only consist of software, the software components described above have to be mapped or integrated with the underlying hardware or rather ECU. This kind of integration concerns the interplay between the basic software and the hardware, so that the basic software can address hardware interfaces and vice versa. Especially for this kind of integration a lot of ECU specific parameters have to be set within the basic software modules and their Templates. During integration, these parameters are at least as important as the interfaces between hardware and software.

- **Software-Hardware-Mechanics Integration (SW-HW-M integration):**

When software is integrated with the hardware, the next step is to integrate hardware with the mechanics. The mechanics is controlled by the hardware. Therefore it is important for integration, that an action intended by a software or a hardware component can be executed by the connected mechanics.

- **Hardware-Hardware integration (HW-HW integration):**

When one ECU and its constituents (software, hardware, mechanics) is working properly, several ECUs have to be put together for building the entire vehicle. The interconnection of different ECUs, each responsible for another job, is realized by a BUS system. This kind of integration has to ensure that the different ECUs, interconnected over more than one BUS systems subdivided into clusters (clusters are interconnected by gateways), work together properly. Working together properly means that ECUs are able to share functionality in spite of the remote communication caused by the distribution of ECUs over the entire system vehicle.

Despite these multiplicity of integration tasks, the following focuses on software-software integration.

2.3.3. Special kinds of integration concerning models and views

After integration and individual kinds of integration within automotive systems were introduced, the following section clarifies the question from section 2.3.1: How does something have to be integrated?

Generally one can say that the thesis addresses integration of components, which were discussed in the last section 2.3.2, on the model level. From this point of view, a **materialized integration** will be performed, as AUTOSAR as well as EAST-ADL2 models particularly concern the design, analysis, or specification level. Indeed it would be possible to use design patterns, like mentioned in section 2.3.1, but patterns are used in the context of executable code. This means, that integration is done at runtime, which slows down the execution time of individual functions and hard real time requirements. Furthermore, integration on the level of executable code can only be done at a late phase of development yet. As implemented modules are a late output of conventional development processes, also problems can only be found at a very late stage.

For this reason and because models are available much earlier than the code, EAST-ADL2 and AUTOSAR enable integration on a model level or rather on a virtual integration. Thereby the basic task of virtual integration is to bring artifacts and language elements of AUTOSAR and/or EAST-ADL2 together. However, integration on a model level can be further subdivided into two kinds of integration, which are of particular interest to this thesis: View Integration and Model Integration. These two integration tasks are detailed now:

Model integration

Caused by the distributed development of an automotive system, a lot of models are created using AUTOSAR or EAST-ADL2. These models are created by even more developers to cover certain aspects of a system in order to analyze and to run interactions among components. This makes it indispensable that models have to be integrated into each other when developers have to communicate or have to work together in a global distributed development process. This means, that model integration is a methodology to construct new models by assembling correlated sub-models or to extract sub-models from already defined models. [32]

Figure 2.9 shows a conceptual view on a model integration. On the model level, multiple models are defined representing different components or properties of an architecture. These models could

be modeled by the same or differing standards, like common UML, AUTOSAR, or... Model integration means putting these models together into an entire system. This makes it possible to analyze or simulate behavior not only of the single models but also of the entire system. For this purpose, the models have to define interfaces or access points. Model or component interfaces inside of the models have to match to ensure that the models or components are able to interact well. Only if models are able to be integrated, an analysis of these models on model level can be performed. For this reason, model integration can also be seen as a virtual integration of components, as system components can be integrated on model level (virtual) already without using expensive hardware or mechanics. Thus, virtual integration can prevent problems during the integration of “real” components on the implementation level, because individual problems and faults can already be found on the levels before.

[34] though distinguishes between a deep integration and a functional integration. While a deep integration produces a single new model which combines two or more given models, functional integration does not yield a new model. Functional integration leaves the given models as they were and superimposes a computational agenda for coordinating calculations over them. While deep integration uses its common formalism or syntax for combining the entire model, the functional integration does not. This work deals with both the deep integration as well as the functional integration. Deep integration concerns the AUTOSAR approach in terms of a common formalism to represent static structures of the entire software architecture on a technical level, the functional integration concerns the usage of other means like an ADL to represent other aspects which rise above the power of AUTOSAR.

View integration

Beside models concerning different parts of the system, there are also multiple abstraction levels or views on that system, which particularly concerns various aspects described by an architecture description language like EAST-ADL2. The IEEE Draft Standard 1471 [36] refers to a view as something that “addresses one or more concerns of a system stakeholder.” Stakeholders are individuals who share concerns or interests in the system (e.g. developers, users, customers, etc.). In this context, a model is the connection of multiple views related to the same system and views are partial descriptions of that model representing information on different abstraction levels (compare to EAST-ADL2 Abstraction Levels 2.2.2) as well as static or dynamic behaviors of a system.

Typical views of (sub-)systems may be the data view as data model, the interface view as black box model, state view as state machine model, process view as workflow model, or the architecture view

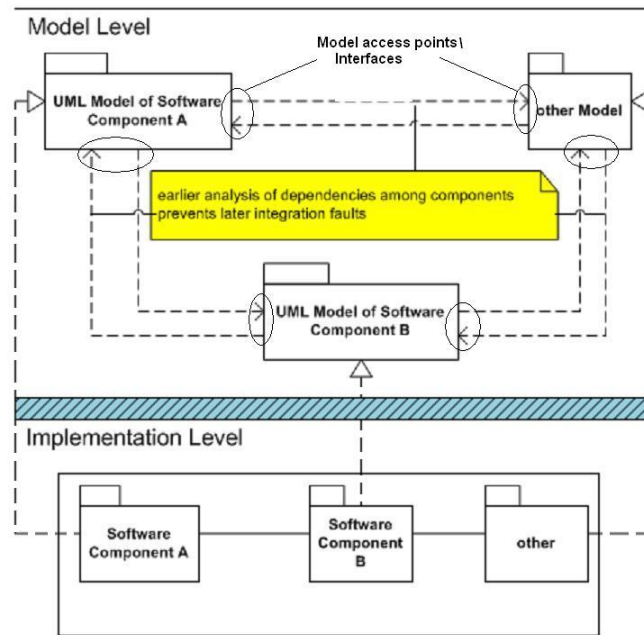


Figure 2.9.: Model Integration (Schematic)

as Architecture model. This means that already small systems mostly consist of a large number of different views so that an integration of them into a model exhibits redundancies and other problems among views. However, since views represent only individual aspects of a system model, those views are meant to be together; only if they are together they can fully describe the model of a system. For this reason, view integration must keep different views consistent.

Thus when using AUTOSAR and EAST-ADL2, it is important to observe correlations between different abstraction levels. However, not only the different views of EAST-ADL2 have to be observed. Especially, transitions between EAST-ADL2 and AUTOSAR views have to be observed, because here the difficulty to integrate two standards complicates view integration.

Figure 2.10 exemplifies view integration by the means of EAST-ADL2 design level view, which abstracts AUTOSAR or rather the EAST-ADL2 implementation level view. The figure shows that there are a lot of “ADLFunctions” on design level, which are refined by further detailed concepts (Runnables) on implementation level. View integration must provide exact relationships between the concepts of different views in order to ensure a continuous transition between the views. This avoids redundancies, indicates which concepts abstracts or refines which component within other views

and enables traceability of design solutions across development phases and stakeholders.

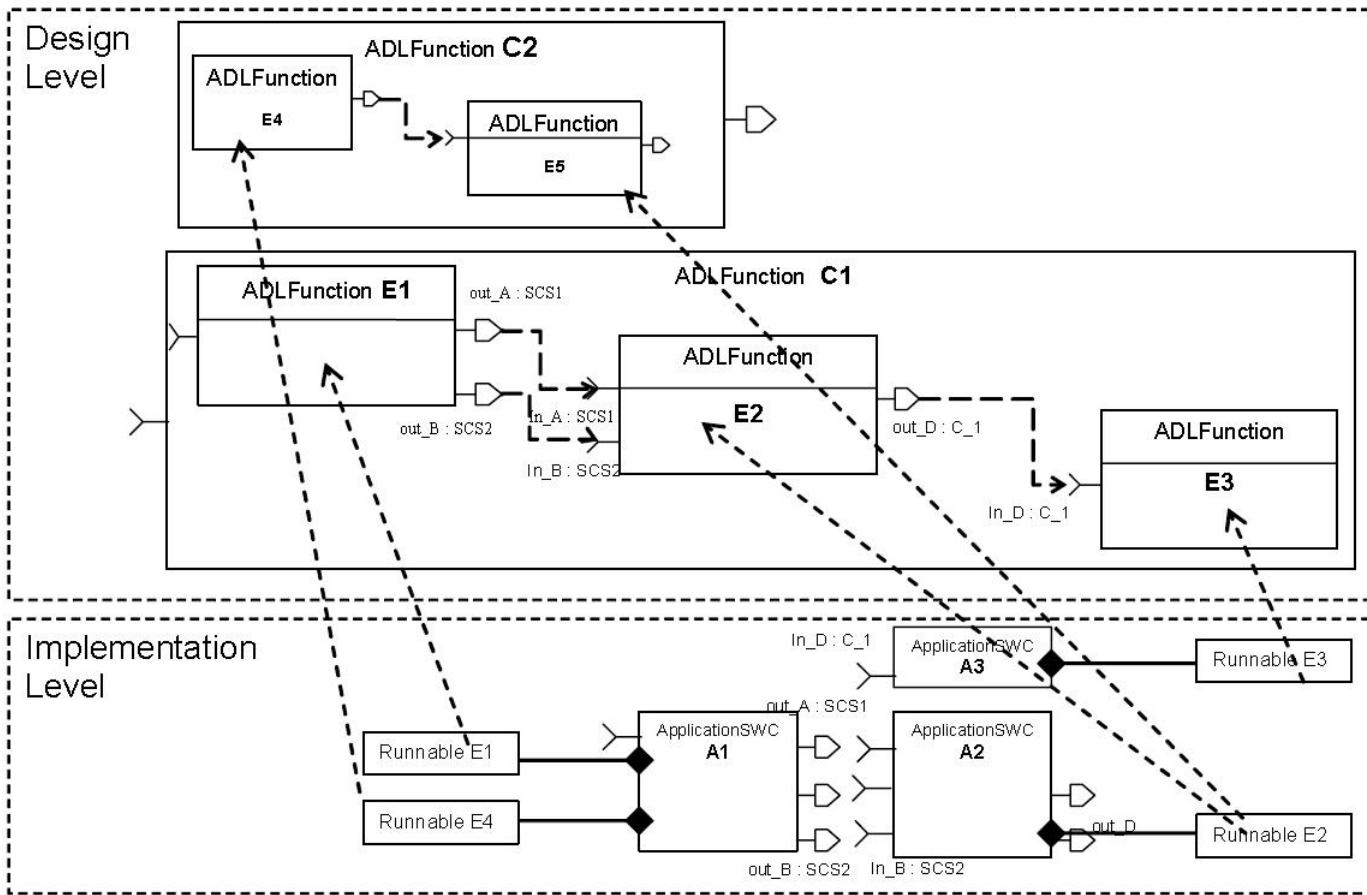


Figure 2.10.: EAST-ADL2.0 Example of a mapping between Design and Implementation Level, from [9]

Prerequisites to integrate models and/or views

The last two sections have introduced the concept of model and view integration. However, there are some most important prerequisites, which must be hold, in order to ensure that integration is possible at all. When using models it is important not only to ensure that components, which are specified by these models, can be integrated, but it is also important to ensure that the models themselves can be integrated. For this purpose it is necessary to ensure that different models are created by holding the same prerequisites, which are detailed by the following listing:

- **Syntax:** Syntax concerns the representation or language elements (compare to section 2.2.1) of models with all of their properties like entities, their attributes, and relationships between

entities, etc. Thus, it specifies the vocabulary as well as model building rules which define what model elements may be connected or not. Furthermore attributes of these elements have to be represented in a standardized way. When models have to be integrated, there must be an agreement of the common syntax between sub-models, which also may be modeled by their own syntax.

- **Semantics:** Matching of data in terms of their semantic is a further task for model integration. This means that there must be an agreement between the semantic or the meaning of model elements represented by the common syntax.
- **Exchange Format:** Another objective of model integration appears in terms of the exchange format of models. This especially concerns the use of different tools supporting the development of them. Many tools provide a mechanism to export data like models. These data are used to be imported by another tool for a further processing. Although XML is a popular exchange format, there is no unique standardized exchange format. The OMG though specifies one possible standard called the XML Metadata Interchange (XMI [48]), but that standard already differs caused by different versions of XMI.

The last sections 2.3.2 and 2.3.3 have answered the two questions from section 2.3.1 for an automotive environment generally. However, covering all aspects of integration concerning an automotive development would blow the scope of this thesis. This thesis wants to confine itself to a certain area of integration inside of vehicles. For this purpose, the following definition wants to restrict integration to software aspects exclusively and should be used for the following sections.

Integration: Integration means combination of functionalities offered by multiple (AUTOSAR) software components (application software and basic software). Thereby an entire system must be built, which provides and manages all functionalities provided from each single component. Unlike conventional integration, which combines real software components in the form of executable code. Virtual integration has to combine models and views, which abstract from real components in order to simplify their development. In both cases, integration means combining static interaction points and dynamic behavior of components, that components are able to collaborate among each other and within their environment.

2.3.4. AUTOSAR integration from a practical point of view

After introducing integration from a theoretical point of view, the following section intends to show different integration tasks from a practical point of view. For this reason a workshop was proceeded to get practical insights into the working with AUTOSAR. The workshop has shown two types of projects, which deal with different aspects of integration. Unlike above, where integration only concerns system components exclusively, other interests must be brought together as well. Therefore the first integration project concerns a common development process for the development and integration of BSW modules. While the second project shows a demonstrator, which exemplifies the interaction of different integrated ECUs whose software is exclusively designed with AUTOSAR. The workshops' goal was to get insights and to find out problems within an AUTOSAR environment. The two projects and the workshops' experiences are described in more detail below:

Integration of BSW Modules: Currently a common process for the integration of BSW modules is under development. The goal is to specify a basic software development process for different and global distributed divisions. Four divisions, each distributed over several locations, presently are developing a total of 53 different AUTOSAR basic software modules. Each division develops certain modules and especially one division is responsible for the integration of these modules. The planned process does not deal with hardware or application. It just concentrates on basic software functionality and wants to coordinate e.g. cross-divisional process phases and roles for BSW module development and integration. For this purpose, the process depends on the common V-Cycle development process and defines phases like planning, analysis, design, implementation, testing, integration, and so on. In parallel with each phase the new process specifies the specification of test cases for certain artifacts. I.e. a developer designs and implements his/her component(s) to define test cases for the developed components.

In addition, the demonstration has used two tools: a configuration tool (Telelogic Synergy) as an example repository for software configurations, specifications and requirements. And furthermore, problems are communicated by a change request tool (Telelogic Change) for gathering and tracing of errors, suggestions, and so on. However, the tools and the distributed development come along with some problems. First of all there is no central database to store all data by now. The Telelogic tools are just examples for tool support. However, there are many other tools and databases in use. This comes along with the drawback that a synchronous data management is not possible in most of the cases. Repository synchronization between Telelogic based systems is realized once a day but this is not the norm. Mostly a configuration or something else is sent over mail and has to be entered

into the own system manually. This missing synchronization may cause multiple faults. For example: if one developer of company 1 makes some changes within component A which depends on a component B. An other developer of company 2 developing component B finds out these changes tardily. The latter developer may make changes within component B causing disastrous effects.

Other problems, which were shown during the demonstration, concern the synchronization of development process phases and the co-development of different modules. A release of an entire block of BSW modules is subdivided into several baselines. These baselines concern individual modules and they define the time when a development phase has to be finished. Furthermore the baselines are points in time, when an integrator has to integrate and test the new (enhanced) modules. Coordination between baselines of different modules and deviations is an important point to avoid delays among different stages of development.

AUTOSAR Demonstrator: Within the scope of AUTOSAR a demonstrator was developed. This demonstrator was constructed over one and a half year to show AUTOSAR functionalities and for learning by doing. The demonstrator consists of three interconnected ECUs concerning three different domains which lead to a cross divisional working group: engine control, cluster instruments, and body. Generally the three ECUs realize an automatic cruise control, a climate control system, and an instruments cluster (tachometer and speedometer). All functions depend on each other. For example: the cruise control system is realized by the engine control system(ECU 1) and depends on the actual speed shown by the instruments cluster(ECU 2). Furthermore, the actual speed has a bearing on the climate control system(ECU 3), which is responsible for heating or cooling of the interior and the engine. All functions within the three ECUs are realized using AUTOSAR. Thus the project is well-suited to show experiences handling with AUTOSAR components. All experiences were captured in so called "Lessons Learned". The most relevant ones are explained below:

- **Tools:** Even a few years ago when the demonstrator was created there were just a few, not mature tools available for developing. This was the first drawback during the development phase. AUTOSAR is a complex framework which can not be handled manually or by insufficient tools not supporting certain parts of AUTOSAR. Today there are much more tools available, but the problems still exist. The complexity of AUTOSAR makes a tool enhancement important. Furthermore, because by now there are a lot of AUTOSAR tools available, interchange between different tools is additional problem.

- **Missing experiences:** Additionally, the missing experiences hamper the development with AUTOSAR. Meanwhile it was learned from the demonstrator project, but experiences need to be gathered further on, in order to specify useful modeling guide lines or best practices, which are enhancing the handling of AUTOSAR and collaboration with other AUTOSAR developers.
- **Missing conventions:** The demonstrator project has shown the importance of implementation rules and clear naming rules as well. Especially for automatically generated codes it is important to prescribe such rules, so that an developer is able to understand that code. Today these rules are partly defined within AUTOSAR.
- **Mapping from AUTOSAR Runnables to OS tasks:** A further problem up to now concerns the mapping of different Runnables onto tasks scheduled by the operating system. The mapping must be implemented manually, because it needs a lot of implicit knowledge about the Runnables to order them. As a consequence of the mass of Runnables within the system this is a very difficult part.

2.4. Inconsistency

Previously, this thesis has described some prerequisites in order to enable model integration as a special kind of integration. These prerequisites come along with some problems if more than one specification of syntax, semantic or exchange format is used for modeling or rather if more than one model is used to represent a large entire system. In other words: when multiple models (or views) have to be integrated, disaccords may occur concerning model elements themselves as well as their relationship (syntax), the content and the meaning of model elements (semantics), and the usage of models within several tools (exchange format). These kinds of disaccords between models disable the possibility to integrate models, which may also have negative bearings on software components. As software components will be derived from these models at the next step of development, problems, which concern the content of models (the system, which has to be developed), can not be found and eliminated on the model level. However, this is exactly the advantage, which is expected from the usage of models.

For this reason, the following chapters will show, that AUTOSAR and EAST-ADL2 provide a common basis for modeling in order to ensure, that models can be integrated generally.

By contrast, not only mismatching prerequisites between models may have negative bearing on the

integration behavior of system components. There are also a lot of reasons, which hamper collaboration between multiple developers generally and independently of any component implementation. These reasons concern the basic condition of development and are responsible for so-called inconsistencies between software components themselves. While the reasons are outlined in section 2.4.1, section 2.4.2 describes concrete inconsistencies in more detail. After that section 2.4.3 describes why it is necessary to avoid inconsistencies (as well as reasons for them) by showing some consequences, which emerge from components' inconsistencies at runtime.

2.4.1. Reasons for inconsistencies: Basic Conditions of Development

The essential artifacts, which must be integrated, are software components. However, in order to develop these software components, there are a lot of divisions, processes, and developers, which have to interact for developing the components. There are a lot of different expertises, languages, habits, or development processes, which come together during the development. For this reason, beside software components, also these different parties must be brought together, that they are able to collaborate easily. Unfortunately, in most cases it is difficult to integrate different parties, that problems, which come along with multiple collaborating parties, have a negative effect on software components too. Therefore the problems, which are sketched in the following, are called the reason for the concrete inconsistencies between software components (see section 2.4.2).

Since there are a lot of thinkable reasons which may be responsible for concrete inconsistencies between software component implementations (compare to section 2.4.2), the following listing just exemplifies some of them. Almost any concrete reason can be classified into a certain class, which aggregates multiple reasons. Furthermore the listed classification of reasons is generic to that extend, that almost any kind of inconsistency between software components can be caused by one of these classes.

1. **Lacking maintenance of artifacts:** Maintenance of artifacts plays an important role in order to ensure the availability of artifacts and information. If developers are not able to orient oneself with multiple artifacts, they cannot find necessary information while they would even be available. However, updates on artifacts must be supported as well in order to ensure that developers work with a valid version of the respective artifact.
2. **Missing Information:** Missing information are another source for inconsistencies between software components on the implementation level. This concerns situations in which informa-

tion is not available, as it had been forgotten or had got lost during the exchange. For example, data get lost if tools use different exchange formats for the import and export of models and information. Furthermore, wrong information is similar to missing information, because in both cases available information is not sufficient to understand specifications of other developers. Therefore it will lead to individual solutions and inconsistencies.

3. **Misunderstandings:** Here, inconsistencies emerge from human and/or technical misunderstandings as well as mismatching specifications. Misunderstandings cause inconsistencies between components when there is more than one developer, even if all components are described exactly. If one developer does not understand necessary artifacts/specifications or components of another developer, it will co-design its own component as if it understands the other one and not like it was intended. On the other side, mismatching specifications concern situations in which two or more artifacts concern the same element. If one artifact describes an element unlike another artifact, a developer is not able to find out, which specification is the right one. Such situations will lead to irritation and misunderstandings, which again may lead to inconsistent implementations.
4. **Inconsistent Processes:** Inconsistencies caused by the used processes concern the cooperation of different teams or companies. When different parties are responsible for different components, which depend on each other, the processes have to be aligned. This concerns the timing of different phases, component releases, the integration of modules, testing of the integrated modules, the communication of information like change requests, and much more.

2.4.2. Categories of Inconsistencies between Software Components

After some reasons for inconsistencies were discussed, this section details the real inconsistencies between software components. Such an inconsistency either emerges from mismatching basic conditions of development, like discussed before, or it comes along with mismatching implementations or models of components directly. To show different kinds of inconsistencies, the following listing based on Saglietti et al. [38] shows a taxonomy of common inconsistencies emerging between vehicles' system components (see section 2.3.2) when they are put together.

1. **Syntactic Inconsistencies:** Syntactic inconsistencies may occur if the data exchange format, by which components communicate, does not conform, e.g. component A sends "String" data to another component B expecting "Integer" data.

However, syntactic inconsistency may also concern model elements like component interfaces, ports, signatures, or data names. For example, if the syntax of an operation of a components' provided port does not match with the operation of the required port of another component, the components can not be interconnected or integrated caused by the inconsistent syntax.

2. **Semantical Inconsistencies:** Semantic inconsistency relates to a differing interpretation of identical data. This happens if data with different or ambiguous semantic are being interchanged between components. Compared to [38] these kinds of inconsistencies may be numerical inconsistencies (e.g. differing number representations), language inconsistencies (e.g. caused by the ambiguity of the language elements), or reference system inconsistencies (e.g. identical physical units differ in semantic, like Celsius and Fahrenheit)
3. **Application-based Inconsistencies:** Application-based inconsistencies are characterized by a violation of states inside of applications or their context inside the system caused by the usage of components without respect to their system environment. That means that components make wrong assumptions about the behavior or the execution semantic of other system components. Constrained service parameter, violation of states saving a legal behavior, violation of obligatory relations between input parameters and internal states, or restricted data ranges may be possible reasons for this kind of inconsistency.
4. **Pragmatic Inconsistencies:** Pragmatic inconsistencies refer to discrepancies concerning the computational environment. Reasons for pragmatistical inconsistencies may arise from concurrency constraints and race conditions between components, access restriction on extern resources, mismatching timing requirements defined by hardware or user interface restrictions. Further risks result from violations of absolute timing requirements or between operations or from violations of relative timing requirements. Furthermore, a mismatching use of communication pattern, like message-oriented or client/server, may lead to inconsistencies among components, thus avoiding a later integration.

In conclusion, the four classes of inconsistency on the one hand concern the vocabulary (syntax) by which components, interfaces and data are described as well as the meaning (semantics) of that vocabulary. On the other hand there are inconsistencies which concern the behavior between components or rather their interplay (application-based). At last there are inconsistencies which concern the components' environment (pragmatic), which does not provide requirements needed by a component.

Inconsistencies between components on model or module level may have extensive effects on integration of components and their later interactions. Therefore the following section describes possible consequences of inconsistencies (section 2.4.2) and their reasons (section 2.4.1), which should be avoided already on model level.

2.4.3. Consequences of inconsistencies for integration

Within this section, individual consequences of inconsistencies are described to show two things: Firstly a couple of faults based on Mariani [43] are described to show the effects, which can be caused by the inconsistencies and reasons from above. Secondly these faults should show the importance of the avoidance of inconsistencies within the development.

1. **Non-Functional Faults:** This kind of fault refers to requirements like reliability, availability, performance, etc.. Non functional faults are particularly caused by application-based or pragmatic inconsistencies. E.g. a component A which must produce a result within x seconds and needs a datum from a component B to complete the computation.
2. **Faulty Interfaces and Ports:** Ports and interfaces indicate what kinds of services are provided or required by components. Normally a components' required port and components provided port must match in order to interconnect two components. This should ensure, that a component can use required functionality, which is provided by an other. However, if there are inconsistencies between such two ports, a component, for example, is not able to use functionality, which is provided by the other component. These kind of fault particularly is caused by syntactical or semantical inconsistencies.
3. **Faulty Connectors:** As components or their interaction points are linked by connectors, faulty connectors emerge from any kind of inconsistency between components. This concerns for example the communication protocol or the data model and message format respectively, e.g. if two components make conflictual hypothesis about the protocol of the connector: (callback versus client-server) or the data format (ASCII stream versus C-based data structures).
4. **Faults on the Infrastructure:** Infrastructure faults emerge from non-conform assumptions of software components and accordingly developers about the underlying infrastructure of the system. These faults are particularly caused by pragmatic inconsistencies, e.g. if components use two different versions of the same dynamic library or if components access the same variable of an distributed shared memory in a wrong manner.

5. **Faulty Behavior:** Even it is possible to interconnect all system components statically, it must be ensured, that components are also able to interact during runtime. For this reason an component must behave like it is expected by other ones. Due to a mismatching behavior it could happen, that a component sends or accepts wrong data. Such wrong data, for example, can result from a wrong message order or a wrong timing, when data are sent or received too late or too early. Thereby a component dos nor only work on wrong data, but it will also return wrong data to other components. Faulty behavior mainly is caused by semantical or application based inconsistencies and could result either in wrong data or in system crashes.

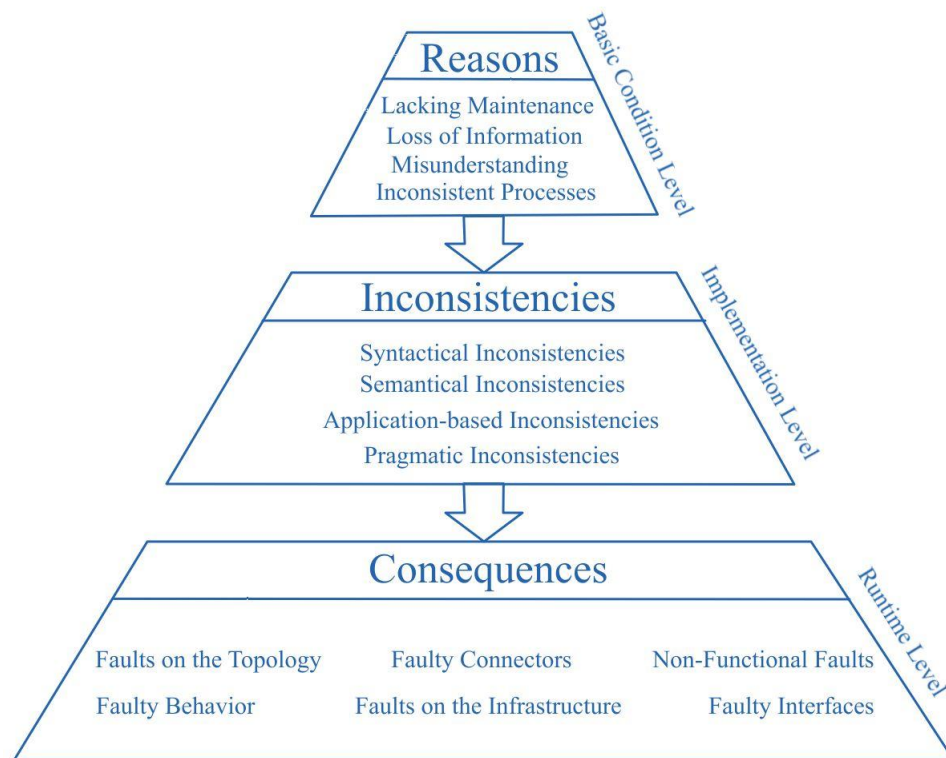


Figure 2.11.: Inconsistencies Overview

After this classification of different reasons, inconsistencies and consequences, which is summarized in figure 2.11, the following definition briefly shows what inconsistency should mean within this thesis and the next sections:

Inconsistency: Inconsistencies are problems between software components hampering or avoiding the static interconnection or the dynamic interaction between software components or their environment. Inconsistencies can occur in the form of syntactical, semantical, application-based, or

pragmatic mismatching. Thereby they are caused from individual reasons concerning the basic conditions of development in most of the cases.

2.5. Related Work

A lot of works deal with the integration of software components into an entire system. They mostly address the same problems and describe possible methods of solution to make a component integration easy. The methods of solution therefore enrich existing component descriptions with additional information, that problems can already be avoided in the forefront of an integration through a better understanding of the component itself and its behavior. Differences between these approaches concern the formalism and the kinds of information which enrich a component description. In the following miscellaneous approaches are exemplary introduced to clarify how integration problems can be tackled.

2.5.1. CCI

CCI stands for Consistent Component Integration. This approach was developed at the Department of Software Engineering of the University of Erlangen and presents an extended interface description language supporting the avoidance and the automatic detection and tolerance of inconsistency classes likely to occur when integrating pre-developed components. This approach focuses on almost all classes of inconsistency as described in section 2.4.2: semantic inconsistencies, application based inconsistencies, and pragmatical inconsistencies, see [38, 39]. To overcome these problems, a strict and formal model was developed to detail components more than usual. Based on this formal model, an UML profile was developed in order to enrich existing UML with additional semantics.

Beside this UML representation, a mechanism was developed in order to generate a Wrapper class for each component. As mentioned above (compare to section 2.3.1) a Wrapper encapsulates components from its environment. The Wrapper gets involved in interaction of components and in other necessary functionalities to monitor and safeguard consistency properties. Additionally, other analytical functionalities, e.g. logging of calls may be provided by the wrapper. By means of the additional semantic provided by the UML Profile, the Wrapper is able to recognize or tolerate possible faults.

More general than AUTOSAR and/or EAST-ADL2, the CCI approach intends to simplify integration by providing model elements with additional information. CCI indeed is not able to describe an individual

architecture, model, or domain. Instead the information are bent on the avoidance of inconsistencies during integration within any domain or model.

2.5.2. UnSCom

UnSCom (Unified Specification of Components) [50, 51] offers a way to describe the external view of a component in all relevant facets. This includes business and technical details which can inform about the expert functionality of the component and of its quality of service capabilities. Each of these levels (or pages, a term borrowed from UDDI [47]) is defined by a meta schema which is composed of the entities available in each level and their relationship to each other. The meta schema also poses restrictions on the cardinalities and introduces properties to the entities as well as value types used to populate some of these. A generic component model forms the basis of UnSCom. This model structures the definition of a component into components, interfaces, connectors and compositions and provides an abstract way to access the interfaces of the component by so called required and provided interfaces. The model adheres strictly to the concept of Design by Contract. Additionally, a type system is introduced, enabling static analysis at design time. Based on the model, a classification schema is proposed, separating the description of the component into three layers, expert, logical (software architecture) and physical (quality). Furthermore, different views are introduced. The static view deals with the structural part of the component, the operational view with the effects and the dynamic view with the interactions of the component. The aforementioned meta schemata for the different pages enable the designer to describe the properties of each page in a standardized way. Together, these descriptions form a complete, structured characterization of the component and therefore a specification framework. Finally, a procedure model assigns the different parts of the component specification to the phases of a development process. The application of the specification framework and an order in which the artifacts should be created is proposed, thus reducing complexity and guiding designers and developers.

By the means of that specification framework it is possible to detail individual concerns of system components on different levels of abstraction/views. Due to the standardized language and some obligatory relationships, which are provided by the meta schema, the components' functionality can be well described on expert level. Thus, misunderstandings and inconsistencies can be prevented on specification level like it is also intended by AUTOSAR and the additional abstraction levels of EAST-ADL2, which particularly focus on the automotive domain.

2.5.3. Other integration platforms

Beside the aforementioned approaches CCI, UnSCom, AUTOSAR, and EAST-ADL2, there are other approaches, whose focus is not on the description of software components itself by using one single standard. Instead, these approaches want to enable the integration and management of different standards, tools, and/or meta models by providing a mediating framework. As mentioned in section 2.3.3, syntax, semantics, and exchange format are important prerequisites, which must be equal between models and/or tools in order to ensure that different models are able to be integrated. However, in the majority of cases different developers and divisions hold different prerequisites. For this reason the platforms and frameworks, which are described in the following, enable the integration of models and tools in spite of differing prerequisites.

- **GeneralStore** is a platform enabling an integrated development process running from models to an executable code, in which heterogeneous CASE tools (e.g., Matlab/Simunk, and AR-TiSAN RT Studio) and their associated code generation facilities are integrated. The software features coupling of subsystems from different modeling domains on model level. From the coupled model it generates a running prototype by code generation.[53] In particular, while using UML models for overall system design, this approach supports the transformations of subsystem models in time-discrete and time-continuous domain. This is achieved by providing meta-level definitions of CASE data in UML, and then integrating the meta models in a MOF (Meta Object Facility) object repository. The data interchange between CASE tools is supported by XML (eXtensible Markup Language). GeneralStore provides support for configuration management. [30]
- **Fujaba** was developed by software engineering researchers for solving the problems of variety of notations and tools in the development process as well as for providing data consistency management in tool integration, particularly at the meta-level. Fujaba supports bi-directional associations between meta-models for different tools that are developed and compiled in separation, such as UML and SDL (Specification and Description Language). To keep models consistent, Fujaba supports automatic consistency checking on model change events. Consistency rules are defined with graph grammar rules. [30]
- **ToolNet** is an integration platform, managing the integration of domain tools targeting specific design phases or aspects of embedded software systems, such as DOORS, Matlab/Simulink, and Borland Together ControlCenter. This approach leaves the domain data at their respective tools. Data integration is achieved by specifying a VirtualObjectSpace in terms of relationships and consistency constraints of domain tool data, which is then stored and managed in a

relation repository. Standardized APIs are used to support tool access and XML based export of tool data. ToolNet provides graphical user interface for navigation. [30]

- **System Weaver** aims to provide configuration management for complex systems. The need is motivated by the insufficiency of traditional document-based specification for large systems. A system information model defines a complete system. Through a defined API or XML file exchange, the platform supports the development of user specific clients (views) and the integration of domain tools. [30]

All these platforms are able to compensate missing standards like AUTOSAR and EAST-ADL2 by integrating multiple quasi-standards, where no common standard exists. However, beside the drawback, which they are not able to describe architectures, models, or domains themselves, these approaches are local solutions of one enterprise or division. They are not able to reach the benefits, which come along with a global standard.

Chapter 3.

Discovering mistakes of integration in practice

In section 2.3.3 and 2.4 essential challenges and problems of an integration of models or common software components are described. Some of these problems may be avoided by the use of AUTOSAR and EAST-ADL2 as well. However, in order to find out what special problems arise at the integration of vehicle software in praxis, some interviews were realized. These interviews are described below.

3.1. A Survey to find out current problems

The interviews should sniff out sources of failures within an accepted software development and their effects on a later integration. On the one hand the interviews' results are used to analyze what kinds of problems of an accepted development can be avoided yet by the use of the AUTOSAR standard and/ or the concepts of an ADL like the EAST-ADL. On the other hand indications for problems which may cause integration faults and which are not enough focused by AUTOSAR or the ADL should be found.

To reach an audience as broad as possible, the interviews were performed in form of a survey whose structure and intention are described in section 3.1.1. The overall survey can be found in the annex of this thesis, see B. After that, section 3.2 summarizes the results of the survey. These results give the focus for this thesis and define its further course.

3.1.1. Structure and goals of the survey

The survey should be fulfilled by an utmost number of people and though constrain the interviewees as few as possible. Therefore it was decided to divide the survey into a mandatory part, which could be fulfilled with 15 minutes, and a voluntary part, which covered some detailed questions in terms of integration.

Inside of the mandatory part the interviewee should be prepared for the topic of integration step by step. On this account, the survey started with very abstract questions concerning the interviewees' working environment and ended with questions concerning the integration. In other words: this part of the survey led from questions about the **basic conditions of development** (compare to section 2.4.1) to questions about **concrete inconsistencies** of integration (compare to section 2.4.2). Overall, the mandatory part of the survey consisted of five blocks, which will now be described:

1. **Role and working environment:** Here the interviewee should give a job description of his daily work. He should describe with what kind of developers or other expertises the interviewee works. The answers of these questions should give information about the perspective (e.g. hardware or software development) from which the survey was fulfilled.
2. **Communication:** Within the communication block the interviewee should describe how he communicates with other developers and what kinds of documents are used for this. Moreover problems occurring in this context should be described. The goal of this questions was to find out general communication problems and reasons for them.
3. **Tool Support:** This block covers the tools/ tool chain, which support developers at their daily work. On this account, the used tooling should be described before problems arising through the use of these tools should be described. The answers were used to find out if the interviewees generally are satisfied with their tool environment or not. A further goal was to get information about the interplay of the different tools and their problems.
4. **Development process:** The development process block asked about the proceeding or process which is used for development. Here the problems concerning the process, its compliance, and the artifacts should be found out.
5. **Integration of different components into a whole:** The last block of the mandatory part concentrated on integration itself. There the point in time was discovered when integration plays a role for the interviewee, what kind of artifacts are important for the integration, and what integration faults occur.

Subsequent to the mandatory, part six detailed questions cover individual and critical problems concerning the integration. There the inconsistencies of section 2.4.2 were concerned particularly:

1. **Artifacts before implementation:** This question particularly covers the relationships and dependencies of artifacts before the implementation. The interviewee should describe possible problems which arise through multiple artifacts, abstraction levels, their dependencies, and the management of these artifacts.
2. **Specification vs. Implementation:** There are mostly a lot of inconsistencies between a specification and its implementation. The question here should give answers for: What kinds of inconsistencies are between specification and implementation? What are the reasons for these inconsistencies?
3. **Semantic inconsistencies between components:** The interviewee should describe particular semantic inconsistencies like defined in section 2.4.2 and such as those arising within his working environment.
4. **Application based inconsistencies between components:** The interviewee should describe particular application based inconsistencies like defined in section 2.4.2 and such as those arising within his working environment.
5. **Pragmatic inconsistencies between components:** The interviewee should describe particular pragmatic inconsistencies like defined in section 2.4.2 and such as those arising within his working environment.
6. **Common questions:** The general questions on the one hand refer to the compliance of Guide Lines and an opportunity is offered for the interviewee on the other hand to address problems which were not covered by this survey or which had not been considered during its creation.

3.1.2. Interpretation of the surveys' answers

The next sections want to format the surveys' answers for the following chapters. For this reason, each question block of the survey is contemplated step by step. First of all, the current situation concerning the respective block, such as communication, tool support, development process, integration or concrete inconsistencies is described. The problems, which come along with the respective actual situation, are then listed and (possibly) subdivided, before their effects on development and integration are derived from them. After that, some results, questions, and problems are outlined. And finally, for each relevant block some possible solutions, which were given from the interviewees for

some referred problems, are listed. A detailed list of all answers can be found in appendix C.

Communication

The current situation in terms of communication looks like this: There are a lot of tools or communication possibilities available and used as well. These tools or means can be subdivided into two categories: communication and data storage. As a “simple” communication most of the interviewees use media, like E-Mail, Phone, or regular meetings. However, video conferences, Wikis, and other internet supported tools, like NetMeeting or WebEx are used as well.

As data storage there are also a lot of tools in use. There are different tools for an exchange of knowledge and project related data, like specifications, work outputs, or code. Examples for such tools are: Telelogic Synergy, Telelogic Change, or Microsoft based repositories called Sharepoints.

Furthermore the survey gave information about the used format by which information are exchanged. It shows that the most of the interviewees prefer textual documents (67,78%). After that, the next mostly used medium is code (19,25%). Models are used very rarely though (9.5%). Other, not further specified exchange formats are used by 7,5% of the interviewees.

In the following, the communication block is subdivided into four problem blocks, which may be concerned by today’s situation: Used Media, Exchange Format, Humans, and Specifications.

- **Communication - Used Media:**
 - **Problems:** The interviewees did not note particular problems concerning the used media themselves.
 - **Effects & Classification:** Consequently there are no effects of these problems, which have to be classified.
 - **Results & Resulting questions:** Although there were no problems mentioned explicitly, the survey shows, that there are a lot of tools and possibilities for communication. This variety holds the risk, that consistency of information between different tools and departments can not be assured. Only this survey does not allow any conclusion whether a consistent data storage is possible and whether data can always be found within a

system containing e.g. more than 1000 documents. But as shown in section 2.3.4, this kind of consistency between tools and locations is still a problem. Thus there is a reason for inconsistency, which can be classified into the reason number 4 (Inconsistent Processes), see section 2.4.1. Inconsistent Processes can emerge any kind of inconsistency mentioned in section 2.4.2.

Furthermore the survey does not give information about particular problems or what kind of medium is the most suitable for certain needs. E.g. is it better to use mail, phone, or a tool like Telelogic Change for a particular information exchange?

A further question, which can not be answered by the survey, concerns the logging of the used media. For example: is it possible to log a medium like phone or video conference? (How) Can logged (informal) information be linked with (formal) documents like models, for which these information are relevant?

- Communication - Exchange Format:

- **Problems:** no particular problems are mentioned concerning the exchange formats' themselves at this point of the survey. The exchange format especially was noted in the "Tool Support point" see below.
- **Effects & Classification:** Consequently there are no effects of these problems, which have to be classified.
- **Results & Resulting questions:** The most used exchange format is textual documentation. Models, on the other hand, are used very rarely. Only 9.5 percent of the interviewees use some kinds of models. However, the survey gives no information about why models are used so rarely. From the point of view of this thesis, the survey shows that models are used insufficiently. Especially because textual documents do not provide any standardized syntax or semantics. Therefore they do not ensure completeness or clearness of specifications like (meta) models do. Caused by incompleteness and ambiguity, which particularly is in the nature of things of textual descriptions without formalism, any inconsistency can emerge from it. However, it has to be clarified if textual documents or implemented modules are better for exchange than other formats like models, or if models are able to solve more problems than they could cause during their introduction (e.g. more design effort, new (non-mature) tools, departure from a well-known development,...).

The following two blocks (Humans and Specifications) and the results of them concern both the Used Media and the exchange format in equal shares.

- Communication - Humans

- **Problems:** Two categories of problems concerning the human factors were mentioned in the scope of the survey: problems with the language and cultural peculiarities of other developers as well as problems concerning the various technical backgrounds and disciplines of these developers.

Linguistic problems refer to the different spoken languages among one or more (global distributed) teams, e.g. German, French, or English. Sometimes this hampers the understanding of descriptions or statements of other developers.

On the other hand, communication problems concern the different cultural and technical backgrounds of multiple developers. Different cultural backgrounds concern the different working styles and habits, which are normal for one certain region or country but sometimes a mystery to other developers from other regions.

By contrast, the different technical backgrounds emerge from different disciplines, expertises, and educations of the individual developers. (E.g. an implementer also has to communicate with a requirements engineer) Even if a developer is an expert on his field, he has to communicate with developers of an other field or expertise.

- **Effects & Classification:** Language problems are not unusual because of the distributed development over several locations within the automotive domain. English indeed is a standard language for development, but it is not the mother tongue of the most. as a result out of that, a lot of misunderstandings are preassigned even if developers exclusively communicate in English. It can also happen that two companies based in different countries use the respective official language for documentation and specification. This may disable developers from reading important documents and especially leads to syntactic (inconsistency number 1 in 2.4.2) and semantical inconsistencies (inconsistency number 2 in 2.4.2). However, other inconsistencies can not be excluded neither.

One has to get used to different cultural backgrounds, but by contrast technical backgrounds are a little more special. Different technical knowledge can disable teamwork or hamper the collaboration among developers at least. This is caused by misunderstandings and the fact that one eventually is not able to understand what another developer demands from him. In addition to the aforementioned possible inconsistencies (syntax and semantics), different technical backgrounds and the resulting problems may lead to application based inconsistencies. For example if a developer does not understand an others' component, he will use the foreign component in a wrong manner.

As an further consequence of language problems and different backgrounds comes a missing communication. Because these problems can reduce communication to a minimum caused by preconceptions. For example: if a developer knows that another one does not understand him (on language or technical level), he will not contact him a priori. Generally, all these problems are caused by reason number three (Misunderstandings), see 2.4.1. However, the problems mentioned above can cause almost any category of inconsistency listed in section 2.4.2. In particular they may cause syntactical (Inconsistency number 1) 2.4.2 and semantic (Inconsistency number 2) 2.4.2 inconsistencies.

- **Results & Resulting questions:** In conclusion it can be said that misunderstandings and global distribution cause a lot of problems. There is a missing transparency between expertises and application fields to link all technical and non-technical information these days. As a result of this, the following questions are important: How can the involved different disciplines and expertises be linked? How can we ensure that all relevant information are available for a developer? How can communication be made more unambiguous? Would an increasing use of alternative media or exchange formats improve the actual situation?

- Communication - Specifications

- **Problems:** The survey shows three problems concerning the development using specifications and other similar artifacts: the documentation of specifications, a missing linkage between these specifications, and the content of a specification itself.

The interviewees note that the documentation of specification and the specifications themselves are incomplete, unclear, and/ or contradictory. In addition, the documentations or specifications may be made inconsistent through changes or updates of them. Moreover they note the difficulty to trace or derive things like requirements caused by a missing linkage between multiple documents and artifacts. Sometimes it is unclear which artifact holds which kind of information. The missing linkage between various system (sub-)components seems to be a problem as well. A further noted problem concerns a general missing or neglect of special kinds of specifications, like for example the specification of non-functional requirements. These kinds of specifications are often neglected by developers (or tool vendors) caused by far too high time pressure or because of some developers think that an executable code is more important than documentation.

- **Effects & Classification:** Based on these problems two main effects result: Firstly, missing information caused by a missing linkage of documents and a missing knowl-

edge about how to find important information. Secondly, misunderstood, misinterpreted or wrong information may be caused by human problems like mentioned above by the missing of information, or by incomplete, unclear, and/ or contradictory specifications. Generally these problems may lead to a development based on wrong information. As a consequence this leads to mismatching connection points, connections, behavior, and wrong assumption about the environment, which leads to any kind of inconsistency mentioned in section 2.4.2. A further problem concerns the discrepancy between specifications and their implementations. This may be caused not only through imprecise working, wrong code generators, or misunderstood specifications. But also through wrongly derived requirements caused by missing specifications or links between specifications and/or requirements.

- **Results & Resulting questions:** Unfortunately, the survey does not give any hints about specific problems with specifications. However, it shows that if data are missing or misunderstood, this may lead to a lot of inconsistencies and faults. Therefore it has to be clarified how documents can be specified in a clear manner. Furthermore it has to be clarified or specified where information can be found and how they depend on each other. In particular, it must be clear which artifact holds which kind of information. Moreover, certain requirements like non-functional ones must not be neglected to ensure that a component fulfills specified and not arbitrary requirements. For traceability it also has to be possible to derive the components' properties from the specified requirements. In conclusion, it has to be ensured, that all specifications are available and complete. At the same time it has to be ensured that they match. For this, the consistency not only has to be ensured for a single artifact, but also between the multiple artifacts. And finally, when all specifications are consistent, it has to be ensured that implementations are close to these specifications. This must be ensured through used code generators and implementers as well.
- The survey does not show whether textual documents and the rare usage of models are the reason for these results.

In conclusion, this block of the survey shows following results and challenges which have to be solved to avoid inconsistencies:

The central question, which results from this block is: How can communication be made more unambiguous? It must be possible to specify all important artifacts, like specifications or documentations, in a complete and unambiguous manner. This also includes non-functional requirements which are important to derive the components' properties from the specified requirements and which must not

be neglected to ensure that a component fulfills specified and not arbitrary requirements. At the same time the consistency of information between different artifacts, tools, and departments has to be assured and information which are communicated over the used communication media (phone, mail,...) have to be logged and linked with the concerned artifacts. The former one guarantees the possibility to integrate all artifacts into a whole, while the latter one avoids repeated inquiring and provides the completeness of information.

Furthermore it must be ensured that all relevant information is available for any developer. For this, it has to be clear or specified where information can be found and how they depend on each other. In particular it must be clear which artifact holds which information. Moreover, the different disciplines must be integrated within the development similarly. For this reason the specifications of different disciplines have to be linked and derived correctly.

In the scope of this thesis it must be clarified if an increasing use of alternative media or exchange formats could improve the actual situation? Are models better than textual descriptions and are they able to solve more problems than they could cause during their introduction (e.g. caused by more design effort, new (non-mature) tools, departure from a well-known development,...)?

Tool Support

The survey shows that there are a lot of tools (not only for communication) in use. Development tools for software specifications, modeling, requirements engineering, simulation, calibration, and programming. Administrative tools for content management, change management, configuration management, and documentation. A complete list of the used tools, which are noted in the survey, can be found in [appendix C](#).

Caused by the large set of tools, in most of the cases it is not standardized which tool has to be used and how these tools have to be integrated into a tool chain.

- **Problems:** The list in [appendix C](#) shows that there are a lot of tools in use. This leads to the first and probably the main problem concerning the tool support: the exchange format. The exchange format enables interaction and collaboration of several tools by providing rules and a format to export information of one tool, which should be imported by another tool. As there is no standard exchange format, multiple individual solutions do not match in most of the cases. This begins already with differing header files of the respective exchange files and ends with differing language elements. Thereby the vendor specific solutions of the exchange formats mostly emerge from a non-applied or non-mature (incomplete) standard.

A further problem concerns differing file structures of artifacts and configurations, which may

differ from division to division or from tool to tool. As there is no overall standard about how to store the files, many adaptations have to be made when multiple files are exchanged between repositories or tools.

Furthermore, the tools enable an automatic code generation to simplify the work of developers by generating some predefined code segments. However, the survey shows that even today's generators are not mature enough for automotive software development.

- **Effects & Classification:** The problems above make a lot of post-processing necessary. Not only file structures have to be adapted and transformed, but also the artifacts themselves and the containing information have to be adapted as well. In the scope of such adaptations it may come to syntactic (Inconsistency number 1 [2.4.2](#) and semantic inconsistencies (Inconsistency number 2 [2.4.2](#)).

Generally, tools have to support developers. But if tools are not able to do their job well and if they are not able to represent important information (modeled perhaps using an other tool), this will lead to some reasons for inconsistency: lacking maintenance of documents created by different tools (Reason number 1 [2.4.1](#)) and misunderstandings (Reason number 3 [2.4.1](#)). All these reasons will lead to any kind of inconsistency.

Especially a non-standardized import/export mechanism (format) may lead to lost information (Reason number 2 [2.4.1](#)) and human misunderstandings (Reason number 3 [2.4.1](#)). This may be caused, for example, by differing vendor adapted meta models inside of the modeling tools. Thereby, too many formats cause differing representation possibilities which hamper understanding of information. Furthermore it is possible that one tool focuses elements more than other ones. Especially non-functional requirements often are neglected, and so these requirements may get lost from one tool to an other. However, if other functional elements, mostly seen as more relevant, got lost from one tool to an other, this will lead to any kind of inconsistency listed in section [2.4.2](#) too.

Furthermore, if an automatic code generation is insufficient, it wastes a lot of resources, like experiences, money, and time, to produce an usable code. Manual reworking of such generated code is very susceptible to human misunderstandings (Reason number 3 [2.4.2](#)), which again may lead to any other kind of inconsistency.

The survey gives no statements about more concrete inconsistencies caused by tools.

- **Results & Resulting questions:** Generally, each single tool which is not integrated into a tool chain does its job well. However, interaction of different tools or the fact that there is more than one tool for the same job within different divisions, slow down the overall development drastically. The survey shows that an standardized exchange format, which is realized and

accepted by each tool vendor in a compatible manner, is needed badly. Defining a general applied file structure for tools, multiple artifacts and configurations is a further important point to support developers at their work, because information can be found more quickly. And finally, better or more mature code generators are needed to minimize manual post-processing. The survey does not show which tool is the best tool for a certain job. However, after a certain training effort each tool may be a good tool. (Perhaps by using well known work-arounds.)

Development Process

Most of the interviewees had noted that they apply the V-Cycle process within their teams. Only one uses a form of RUP, the Rational Unified Process. The problems concerning the process and their consequences have to be described below.

- **Problems:** There are five main problems, which were noted in the scope of this survey. (1) The project schedule mostly is not detailed enough and at the same time it is not well communicated. (2) Because there are a lot of people involved in development, there are also a lot of different interests and targets between the teams. (3) On the other hand responsibilities (defining who has to do what within the process) are unclear defined and (4) it is difficult to contact developers, who are responsible for other (former) projects and modules. (5) A further problem concerning today's development processes seems to be the missing or neglected risk management and escalation procedures, which are defined only rarely.
- **Effects & Classification:** (1) Caused by the restricted project schedules and the restricted process appliance, some important steps for documentation and a clear specification are neglected. (2) The differing targets and interests emerge competition/ concurrency and other focuses on quality among different teams. This may lead to situations, where the teams lose sight of the actual target. They are busy doing standing up for their interests. (3) No clear defined roles and responsibilities make it unclear who has to do what for certain process phases. As a consequence it is not clear who has to be contacted when problems or questions arise. This leads to point (4) where missing contact information may lead to a situation in which developers are not able to communicate with other involved developers. In this case they simply do not know who can be contacted. (5) A missing or insufficient risk management concerns undesired situations within the development process. If there is no risk management or escalation procedure defined or prepared, it is too late to define strategies when the disaster has happened. Such a missing firstly causes a lot of development time and much more money secondly.

These resulting kinds of problems emerge from reason number 4 (inconsistent processes), which may lead to any kind of inconsistency of section 2.4.2. The survey gives no statements about concrete inconsistencies caused by inconsistent processes.

- **Results & Resulting questions:** In conclusion it can be said that there are a lot of deficits in terms of the development processes and particularly in terms of a cross-divisional process. Sometimes a too restricted schedule causes an imprecise proceeding and the vague definition of roles and responsibilities causes lacks of clarity concerning the application fields and contact persons. Furthermore, a missing risk management does not compensate situations in which the mentioned problems occur. Therefore it has to be analyzed if an overall process, which purports and contracts more than today's processes, could solve these problems. Sadly, the survey does not show how tests are involved in today's processes. However because of tests are an essential part of development, they must not be neglected in terms of an old or newly defined process.

As described in section 2.3.4 there is attempt to define a common cross-divisional process for BSW module integration these days. For this reason, a cross-divisional development process is a point, which is not further contemplated. Experiences and results of that process should be taken into account before defining further cross-divisional processes.

Integration as a whole

The survey shows that today's integration as described in section 2.3 primarily uses implemented modules or textual descriptions. Also models and interface descriptions are in use rarely. In terms of models and the integration on model level there are just two oppositional opinions: yes or no. Some of the interviewees think that models and their integration can be helpful as models enable an early system overview and the possibility to define and simulate test cases. The other party thinks that models are not helpful as most of the problems are handled on other levels like code and the tools are not mature enough. In spite of these negative opinions in terms of models, a clear trend can be recognized. Most of the interviewees expect a big improvement from the usage of models and the possibility to integrate multiple (sub-)models into an entire model. After the actual situation concerning integration was described, the problems which hamper integration and their effects have to be detailed. These problems are used as basis for further problems, which will be concretized within the survey's optional part.

- **Integration faults and their reasons:** As often mentioned insufficient and wrong specifications are a big reason for a lot of inconsistencies. This also hampers the integration of

components, which are not well-described. Imprecise, inconsistent, or missing information may be caused by not well-communicated component or specification changes but also by an insufficient communication between developers. Changes may concern modules themselves, used libraries, or the like.

Beside the difficulty to overview the complexity of an entire system, which is a further problem, mismatching component interfaces are also a main problem for integration.

Furthermore it is important for integration to know certain details about the environment of the component, which has to be integrated. The survey shows that even such information are missing. Environment here does not only mean the software environment of a component but also for example the real environment of the hardware, which executes the software component, and the real world, which also influences the software (or hardware).

There are also different variants of software or hardware components for differing variants of OEMs and vehicles. This makes it necessary to manage these differing variants even for integration tasks. However, the survey shows that today's variant management is insufficient or unavailable.

- **Effects:** Generally, integration problems are caused by insufficient or wrong information about components. Reasons for integration faults therefore are a lack of maintenance (Reason number 1) or misunderstandings (Reason number 3).

This leads to situations in which components are not able to be interconnected. Situations like this may be caused by syntactic (Inconsistency number 1 [2.4.2](#)) or pragmatical (Inconsistency number 4 [2.4.2](#)) inconsistencies. However, if components are able to be interconnected, it does not imply that components are able to interact. Due to aforementioned missing or insufficient information it is possible that components' behaviors does not match. Data are sent or computed in a manner, which can not be handled by involved components. This may be caused by semantic (Inconsistency number 2 [2.4.2](#)) or application-based (Inconsistency number 3 [2.4.2](#)) inconsistencies.

The disability to manage and integrate different variants of a component emerges from the fact that it is not well-supported by actual standards. However, due to the large set of variants, which are generated during the development of multiple components for multiple OEMs and vehicles, it is also important to support the management of them. Because if variants are not well supported, it can happen that an engine control system of a vehicle with 40 hp controls an engine with 230 hp. Problems would be preassigned. More details about concrete faults are not given at this point of the survey but further below.

- **Results & Resulting questions:** The survey shows that future means have to help reduc-

ing complexity of entire systems and supporting the management of variants. Furthermore, the interfaces or rather the system have to be specified in an unambiguous, complete, and standardized manner. For this reason the semantics behind interfaces or data, the behavior of components and their environment have to be described as well. In the case of changes it must be possible to be communicate these changes well.

- **Suggestions (from the interviewees):** In the case of changes, the following information have to be recorded for each change and component: (1) What is new compared to other baselines and builds? (2) What has changed compared to baselines and builds? (3) What dependencies to other modules or baselines are concerned by a change? (4) What fault was resolved by a change? (5) Who made a change?

Concrete inconsistencies

All the points from above concern particular abstract problems or the basic conditions (see section 2.4.1) of an automotive development which may cause any kind of inconsistency. The obligatory part of the survey does not show detailed faults which can be categorized into an individual kind of inconsistency. But within the optional part some detailed information about concrete inconsistencies (see section 2.4.2) were noted.

1. **Integration and Syntactic based problems:** Especially the interfaces between components are concerned by syntactic inconsistencies, which hamper the interconnection of components. If syntax between interfaces does not even match, it also not possible for components to interact. This means to ensure that any kinds of interfaces are describable on syntactical level. The descriptions have to be complete and unambiguous as well.
2. **Integration and Semantic based problems:** Data, which have to be exchanged between components, are often inconsistent caused by mismatching semantics. This means that a component misinterprets data received from another component. For example, if one component sends a data type float with the semantic of speed in kilometer per hour and the receiver component misinterprets this float with the semantic of speed in miles per hour. Beside data the semantic of interface descriptions also may be unclear. This does not only concern the name of interfaces, but also the job (What does an operation compute?) and the behavior (How to use the operation? Is there an operation sequence order defined which must be hold?) which is represented by an interface. This means that the semantics of data and operations has to be well-defined.

3. **Integration and Application based problems:** On application level it was noted that values received by a component are even obsolete when they are processed. This means that it must be ensured that an component uses values within a predefined time or that an value has information about when it is obsolete.

The survey does not give any hints about problems concerning interaction protocols and the states of a component.

4. **Integration and pragmatic based problems:** Pragmatic inconsistencies emerge from unobserved or unspecified timing restrictions of one or more interacting components. This either concerns the minimal or maximal runtime of a single operation or the runtime of multiple operations, which are executed successively. So the timing behavior of components has to be specified in a formal and obligatory way.

Furthermore it has to be ensured that an component is used within the right context of the system. This concerns the performance of the underlying hardware and environmental software components as well.

However, as there is an actual project called TIMMO (Timing Model), which particularly addresses the specification of components' timing behavior, this thesis will not further focus problems concerning the timing.

3.2. Results of the whole survey and further course

Before the thesis one thought that the survey will show some concrete inconsistencies between software components caused by the actual development style. In a second step it should be analyzed whether these inconsistencies can be avoided by the support of AUTOSAR and EAST-ADL2.

However, the survey shows that problems do not concern a concrete level of particular inconsistencies, as described in section 2.4.2. But more general problems concerning the communication, specifications, development processes, or tools are responsible reasons, which can lead to the described concrete inconsistencies, like described in section 2.4.1. It makes it even more obvious that a lot of information was given within the obligatory part of the survey, which concerns the abstract level more than concrete inconsistencies while within the surveys' optional part, which particularly concerns concrete inconsistencies between (software) components, just a few problems were noted. Even within the obligatory block "Integration of different components into a whole" just few concrete inconsistencies, which hamper integration, were noted.

Therefore one can say that concrete inconsistencies between components do not avoid integration predominantly. However, basic conditions of development cause the problems, which affect modules and their integration possibility in a second step. Thus, the thesis' target must be realigned at this point of the thesis:

All present reasons for later inconsistencies on implementation level, which come along with developments' basic conditions, must be analyzed. For this reason section 3.2.1 summarizes the most important results from the survey, which particularly concern problems of basic conditions of development. Afterward, section 3.2.2 summarizes the surveys' results particularly for concrete inconsistencies. The problem classes, which are identified during this summarization are taken on section 4.1 in order to analyze these listed critical points of the actual development style. There the analysis has to find out, which enhancements or advancements come along with the new development paradigm using AUTOSAR or EAST-ADL2.

3.2.1. Reasons for inconsistencies

This section describes the surveys' results, which particularly concern the basic conditions of development (compare to section 2.4.1). The results from the respective survey blocks (compare to section 3.1.2) are categorized into problem classes, which will be analyzed in the context of AUTOSAR, see section 4.1.2, and EAST-ADL2, see section 4.1.4.

1. **Benefits from the usage of models:** Generally, as the survey shows that models are used rarely, it must be clarified whether models are better than other formats like textual descriptions. It has to be analyzed if models are able to reduce the complexity of a system to avoid problems of a conventional development (see also section 3.1) more than textual descriptions or code itself.
2. **Completeness & Clearness:** It must be possible to specify all important artifacts, like specifications or documentations, in a complete and unambiguous manner. However, not only the completeness and unambiguity of a single artifact has to be ensured. Also the whole set of specified artifacts must be available and dependencies among them and the content of the respective artifact must be clear. This also includes that individual abstraction levels for analysis or design which contain, for example, (non-)functional requirements and variants, have to be involved.
3. **Consistency:** Consistency of information or data has to be ensured, whereas consistency must be hold between three different elements. 1. Between artifacts' data, which may be

defined on the same or different abstraction levels. This means that it must be possible to derive and update information between artifacts on different abstraction levels correctly. Also artifacts on same abstraction level must be hold consistent. This concerns, for example, interfaces and data, which are defined and related within more than one artifact. **2.** Consistency must be hold between tools, when information is exchanged between tools. It must be ensured that exported information of a tool A conforms to the imported information of a tool B. For this, a standardized exchange format, which is realized and accepted by each tool vendor in a compatible manner has to be established. Moreover, it must be ensured that an import and export of data even is possible without adaptations or work-arounds. **3.** As artifacts also have to be exchanged among several departments and division it must be ensured that each party works with a valid and actual artifact. This point also depends on one of the following points, "Maintenance of data".

4. **Continuity & Traceability:** Beside a whole set of artifacts, it must also be ensured that developers or stakeholders are able to switch between all artifacts continuously. This should support the availability of information not only of one single artifact, but also from other points of views, concerns, or components. Furthermore, concerning different abstraction levels and respective artifacts it must ensure traceability of individual design solutions among several abstractions, process phases, and disciplines.
5. **Integration of all disciplines** Because there are more than one discipline involved with development of vehicles (e.g. Requirements Engineer, Software Architect, Implementer,...), it must be possible to integrate different disciplines into the development in equal shares. This means that information tracing between different disciplines must be possible to understand influences of other disciplines on the own.
6. **Maintenance of artifacts:** To reduce the complexity of the large set of artifacts, which are created during development phase, it must be possible to manage artifacts in a standardized and well-known way. On the one hand this means that a general, well-known project file structure for (different) tools and configurations must be applied. This should ensure that developers are able to find relevant information inside of each configuration management system, model repository, or something else. On the other hand it must be possible to manage different artifact versions, baselines, or releases. For this reason it must be possible communicate changes to other developers. This is an important point, because changes within one component may have influences on the behavior or the development of other components caused by dependencies between components. It must be possible to annotate the kind of change, the impact of that change on other components, or the responsible contact person of that change

as well.

7. **Assignment of roles and responsibilities:** In order to specify who is responsible for which task, it is important to specify roles and their responsibilities. It should be clearly defined, who has to do a particular task within the development process or not. Beside this definition, roles enhance the possibility to contact responsible persons to clarify possible problems. For the latter advantage it should be possible to annotate detailed contact information to other developers or projects for each specification, module, or information.

3.2.2. Concrete inconsistencies

This section describes the surveys' results, which particularly concern the concrete inconsistencies (compare to section 2.4.2). There the results from the respective survey blocks (compare to section 3.1.2) are categorized into problem classes, which will be analyzed in the context of AUTOSAR, see section 4.1.3, and EAST-ADL2, see section 4.1.5.

1. **Static properties of interfaces:** Beside the demand to specify entire system architecture in a complete and unambiguous manner, component interfaces play a major role for integration, as they represent the interaction points between components. In order to integrate software components on module level, the syntax of interfaces has to match. As concrete interfaces should be derived from their specifications/models, the specification level yet must provide the syntax to specify interfaces in an unambiguous, complete, and standardized manner. This ensures either that modules properly can be derived or generated from specification level above, or that specifications/models can be integrated on design/ model level yet.
2. **Dynamic properties of interfaces** When components can be integrated on specification or module level, it does not mean that components are also able to interact. To reach this nevertheless, it must also be possible to specify the semantics behind interfaces or data, the behavior of components, and their environment. It must be ensured that all important properties, which form the dynamics of an application, can be defined by specifications/models.

All these points must be analyzed within the following section 4.

Chapter 4.

Solving the actual problems using AUTOSAR, EAST-ADL2 and additional concepts

After identifying the most critical points of the current development style for automotive software, the following section will analyze these points in more detail. Therefore section [4.1](#) shows how far AUTOSAR in combination with EAST-ADL2 is able firstly to enhance the actual basic conditions and secondly to avoid concrete inconsistencies by the means of their meta models and specifications. Afterward, section [4.2](#) sketches an approach, which completes AUTOSAR and EAST-ADL2 with missing language elements and information.

4.1. Analysis of the surveys' results

AUTOSAR and EAST-ADL2 are model based solutions for specification of large automotive software architectures. However, the surveys shows that models are used rarely. In order to warrant the use of AUTOSAR and EAST-ADL2 models, section [4.1.1](#) will clarify whether models bring advantages compared to todays' applied formats generally. After that, section [4.1.2](#) and section [4.1.4](#) will detail how far AUTOSAR and EAST-ADL2 are able to enhance the problems of basic conditions, which were identified in section [3.2.1](#), while section [4.1.3](#) and section [4.1.5](#) discuss AUTOSAR and EAST-ADL2 in terms of concrete inconsistencies, which were identified in section [3.2.2](#). Finally section [4.1.6](#) summarizes how far the actual development can be enhanced or supported by AUTOSAR and EAST-ADL2.

4.1.1. Why are models better than textual documents or executable code?

The following section will compare models with textual documents and implemented code to show advantages and disadvantages of the respective format. Particular properties, which should be achieved by any format, are compared as well as particular possibilities, which come along with the respective format.

Format Distinction: Syntax Support

When information is exchanged, it is important to understand the representation format of information. In the case of textual documents, it is not very difficult to read them when someone is familiar with the language which is used inside of these documents. When models are used, it is not so difficult to understand them too, because models abstract from finenesses and provide a graphical and clear syntax. A graphical syntax simplifies understandability even more, because graphics mostly are better to understand than other formats like native text or code. When the modeling syntax is well-understood, models will show dependencies between model elements and will provide a good overview of the specified system. By contrast, the implemented code indeed also provides a clear syntax, but on the other side it is hard to understand. Mostly it is very difficult, even if someone knows the programming language, to understand dependencies between variables, functions or classes if the code has not written by oneself.

Format Distinction: Semantics Support

Understanding the representation format of information is one thing, but one also has to understand the meaning of the represented information. Textual documents indeed are less time-consuming, but the survey shows that they are affected by many misunderstandings caused by linguistic problems and their incomplete, contradictory, and informal character. This causes a wide room for interpretation of this kind of specification. Implemented and executable code in contrast is very time-consuming and affected by misunderstandings as well. Implementations indeed are nearly complete and formal, but it is very difficult to understand the functionality of the code just using the code itself. For this reason, the implemented code uses comments for documentation in form of text to clarify the functionality of functions, variables, and classes. However, this again causes the same problems as described for textual documents. On the other side, models are concerned by misunderstandings rarely. Meta models, profiles, and the possibility to define constraints ensure a predefined,

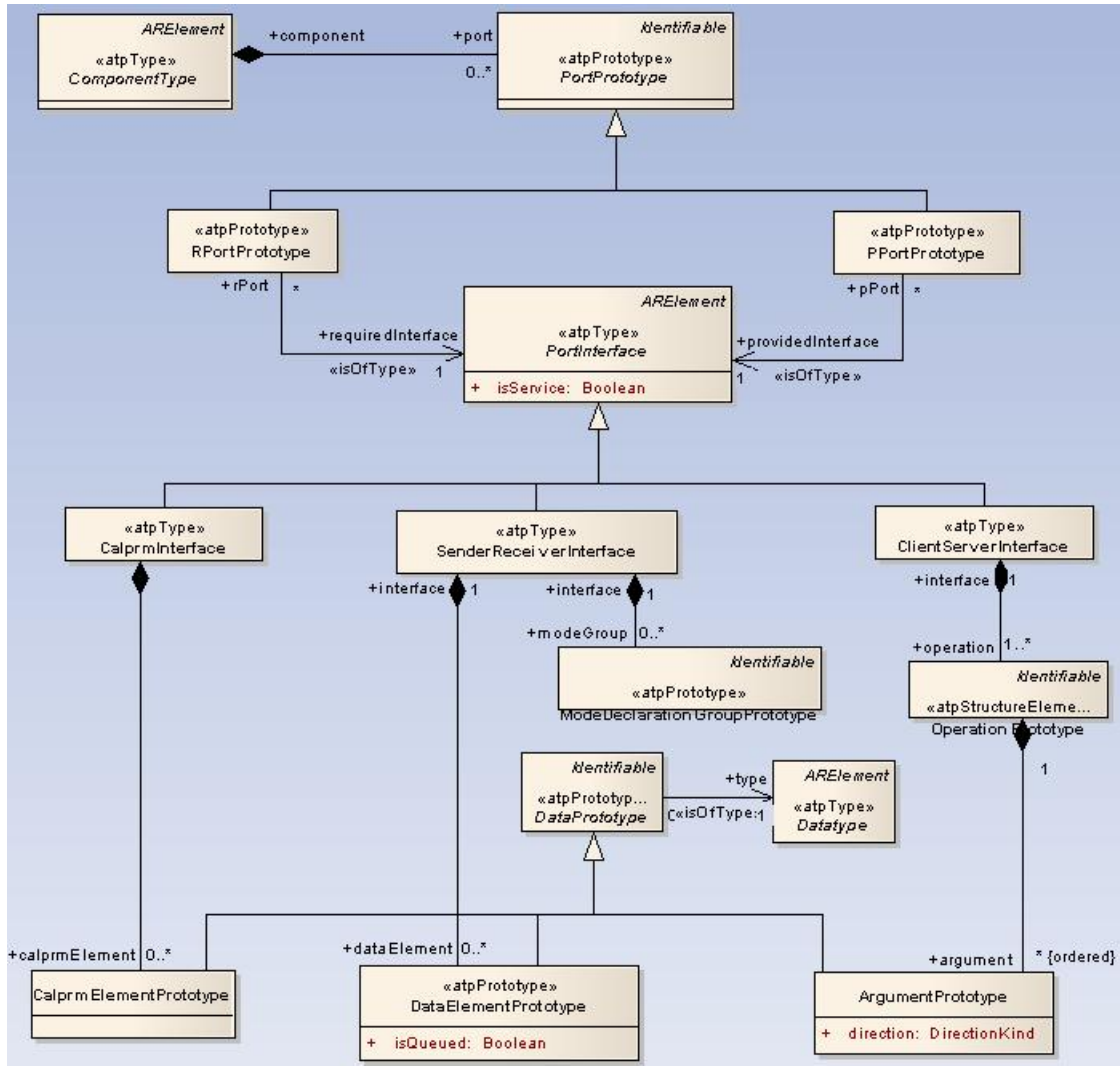


Figure 4.1.: AUTOSAR Components-Ports-Interfaces, from [12]

maximal necessary syntax in a semi-formal manner. Meta models ensure the completeness of the derived models by prescribing important or obligatory relationships among model elements, necessary variables, and constraints for the usage of model elements and variables on model level. In this context, meta models provide abstract syntax and semantics as well, which can or must be applied on model level. It is easy to understand models, if the comparative small syntax of the meta model is understood. After practice and when the modeling technique is understood, modeling is not very time-consuming, because it can be seen as fulfillment of predefined model elements and their relationships.

Format Distinction: Consistency Support

A further problem of specification in form of textual documents is, that implementations are not consistent with their specification in most of the cases. This is caused by the very informal character of textual documents, which enables a lot of interpretations. This informal character practically makes it impossible to transform textual documents into another format or to generate code from text automatically, while models are suitable for transformations and code generations optimally. These days', model transformations, which transform models into other models or representations, are no problem using languages like QVT (Query View Transformation), which was specified by the OMG [49]. And as a particular kind of transformation, a lot of mature frameworks like OpenArchitectureWare[2] or the Eclipse Modeling Framework[1] provide automatic code generation for models as well. The advantage is that transformations as well as code generations can be repeated as often as necessary. Thus it is possible to update model modifications within dependent models or implementations automatically.

An automatic consistency check of textual documents is practically impossible because native language can not be understood by programs, which need strict formal rules to interpret inputs. In the scope of implemented code in contrast, it is possible to use compiler or model analyzer [31] for checking whether the content of code conforms to the syntax of the programming language and whether dependencies between variables or functions are consistent. Furthermore it is possible to debug code in progress to find and to understand faults. Models provide the possibility of consistency check as well. There are also model checkers/analyzers like ATL (Atlas Transformation Language)[29] which assert the well-formedness of models and the compliance of a given model to its meta model. On the other hand, models are able to be simulated, but the possibility to debug these simulation is very restricted and not mature enough these days.

Format Distinction: Continuity Support

Two last advantages which come along with models more than with implementations or textual documents concern a later adaptation of language elements on meta level or below and the traceability between different artifacts. While changes within textual documents or implementations cause a lot of manual reworking within other documents or implementations, changes or adaptations inside of models can be propagated simply. Changes on model level, which affect dependent models or model elements, can be updated easily, because all model elements are interconnected via relationships,

which indicate dependencies between them.

For example: a model element A defines a variable, which is used by another element B. If someone changes the variable of component A, the used dependencies between the two components, indicates, that also the used variable of component B must change.

Furthermore, model usage even enables to change the meta models' abstract syntax and semantics ex post in order to adapt language elements on model level. By the means of model transformers it is possible to update changes on meta level, also on model level automatically. In an analogous manner it is possible to update changes on model level, also on implementation level by the use of code generators.

Integrated development environments (IDEs) indeed enable similar adaptations on code level as well, but not for specifications which are also be covered by models. An adaptation of language elements of the programming language itself is impossible too.

Format Distinction: Traceability Support

On the other side, textual documents and implementations do not provide mechanisms to trace between abstraction levels of specifications, (sub-)systems, or artifacts as simple as models. One model, however, can consist of a lot of sub-models which are interconnected by multiple specified relationships like refinements or dependencies. Thus, developers can trace within one model between several models, abstraction levels (views), and information. While textual documentations provide this functionality only with the additional use of configuration management systems, implementations only provide this on code level or again with the additional usage of configuration management systems. The disadvantage of such configuration management systems is that the large set of documents and the missing graphical representations of dependencies among artifacts make it difficult to keep an overview on all documents and to find the relevant artifact quickly.

All in all it seems that models provide more advantages than disadvantages in comparison with textual documents and the implemented code. Probably they are used so rarely because they are new territory for the most of the developers. However, as summarized in table 4.1 models come along with enough benefits, which warrant the use of AUTOSAR and EAST-ADL2.0 in the following. The future will show if models completely enter the work of automotive developers as within other domains.

	Text	Code	Models
Usage	often	often	rare
Understandability	understandable	difficult	well understandable
Misunderstandings	often	often	rare
Completeness	mostly not	yes	mostly
Formalism	no	yes	semi-formal
Time expenditure	low	high	low
Close to implementation	no	yes	mostly
Code generation	not possible	not necessary	possible
Consistency checks	not possible	difficult	possible
Transformations	difficult	possible	simple
Adaptations	difficult	difficult	simple
Traceability	difficult	difficult	simple
Debugging	not possible	simple	not mature

Table 4.1.: Comparison between models, textual documents, and code

4.1.2. Basic Conditions & AUTOSAR

Some reasons for inconsistencies (compare to section 2.4.1) could be avoided by using existing AUTOSAR concepts. If the AUTOSAR standard would be applied by any OEM or supplier, the common base could avoid most of the problems, faults, or failures. On the basis of the points, which were identified in section 3.2.1, the following will show how far AUTOSAR is able to enhance the actual situation, which was criticized within the survey.

Completeness & Clearness

The actual AUTOSAR standard intends to enable the description of automotive systems just on a technical level. It consists of 139 structured documents which contain 7629 pages of textual documentation and the AUTOSAR meta model. About 140 mega bytes result from this set of documents, whose rough structure is subdivided into five parts, which can be found at [10]: a “Main” part, “Application Interfaces”, “SW Architecture”, “Methodology & Tools”, and “Conformance Testing”. Furthermore, many secondary literature can be found on the internet. Alone this large set of specifications could lead to the assumption that AUTOSAR must be complete. However, the quantity of documents does not give any hints for their quality or completeness. While completeness is important, as missing artifacts will lead to individual solutions or work-arounds, the quality or clearness ensures

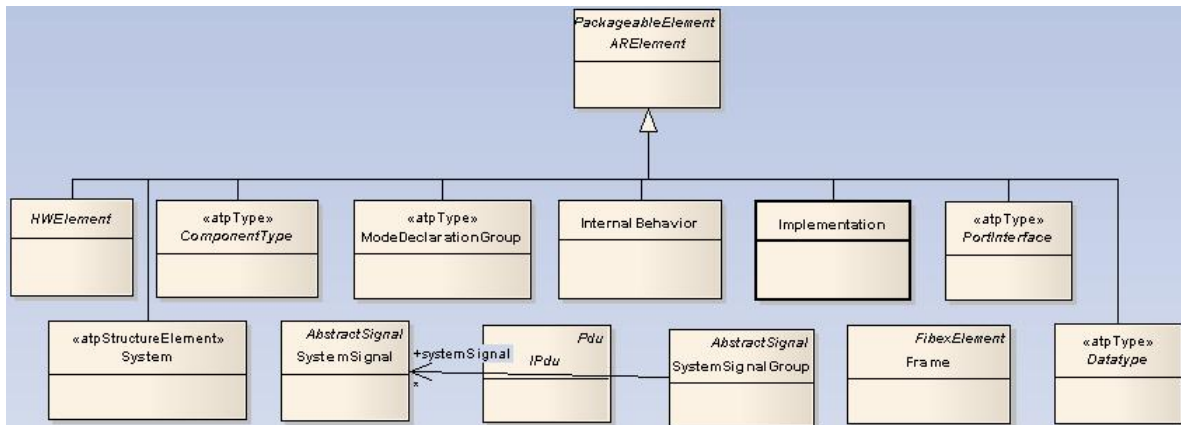


Figure 4.2.: AUTOSAR Elements, from [12]

understandability of specifications.

For modeling automotive architectures AUTOSAR provides a meta model which model elements can be used for specification on model level. For this reason, the meta model as well as a model as instance of that meta model must be complete and clear. The meta model must provide a complete and clear set of model or language elements, which are necessary to specify all aspects of automotive architectures. Also on model level it must be ensured that the model of a concrete architecture is complete and clear. Hence the two levels must be distinguished for completeness and clearness in the following.

On meta model level

- Completeness on meta model level:** The meta model must be complete in terms that all syntactic elements, which are necessary to describe a system on a model level, are available. If the abstract syntax of the AUTOSAR meta model does not provide all necessary syntax elements, it will not be possible to describe an AUTOSAR model properly. However, for larger meta models it is hardly possible to verify that the meta model provides all domain specific elements. This comes from the fact that meta models are defined by domain specialists, which possess a large knowledge about the respective domain, manually. Due to the importance of detailed domain knowledge, which no machine can possess, it can not be automated. And because of the complexity of automotive systems it can not be ensured that even a domain specialist forgets some details.

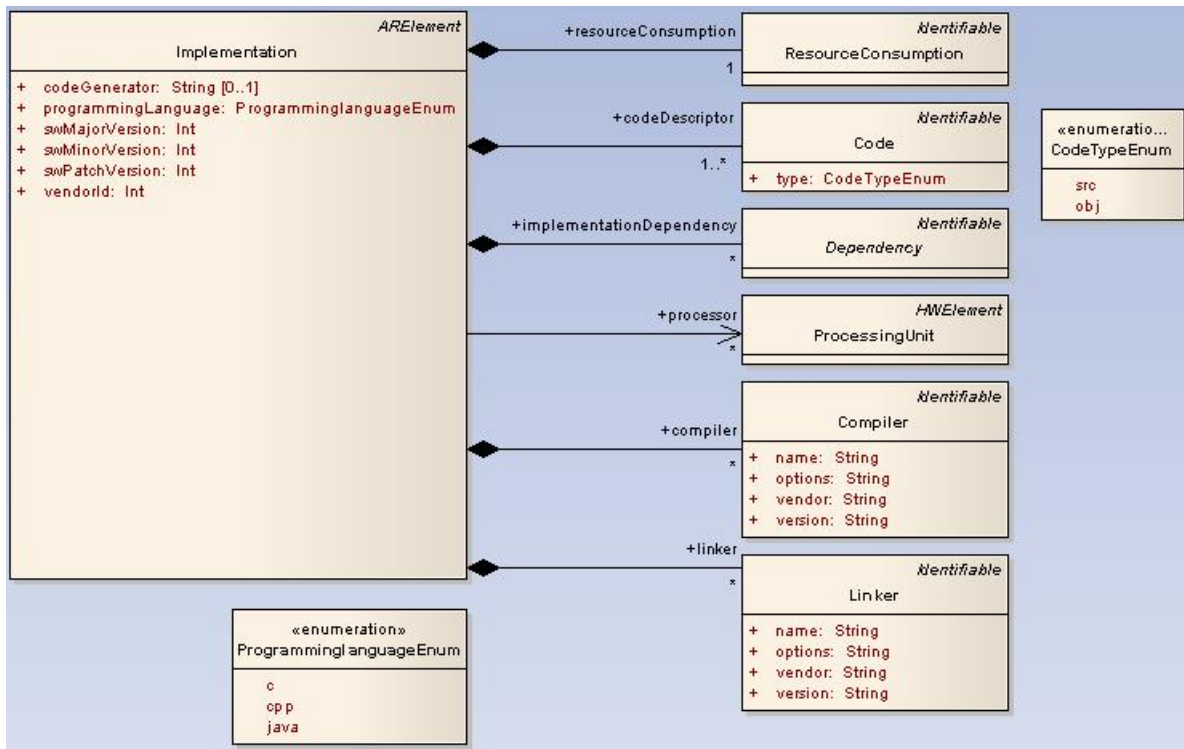


Figure 4.3.: AUTOSAR Software component implementation, from [12]

Closed projects have shown some missing concepts or elements, which were entered in revised AUTOSAR releases. However, by now it can not be said surely or verified that the actual version of the AUTOSAR meta model is complete or incomplete. At this point, only a rough overview about the meta model must suffice for the assumption, that it is able to describe automotive systems on the technical level completely. Incompleteness will be shown by further test projects in the future.

Figure 2.3 shows the overall meta model or template structure of AUTOSAR. One can see that AUTOSAR describes software components(basic software and application software), hardware components(ECUs), dependencies between these components in terms of a system, some generic ECU specific parameters, and other overall used elements. Furthermore, figure 4.2 depicts all AUTOSAR elements, which are used within these just mentioned elements to describe elements like HWElements, ComponentTypes, InternalBehavior, Implementation, PortInterface, DataType, or SystemSignal.

“HWElements” (hardware) can be an ECU, a memory segment, a peripheral, a processing

unit, a sensor, an actuator, or other ECU electronics. “ComponentTypes” represent software components for applications, sensors, actuators, ECU abstractions, drivers, services in terms of the basic software, or a composition of such software components. “Internal-Behavior” and “Implementation” classes provide detailed information about the “Component-Types’ ” “Behavior” or “Implementation” of software components. PortInterfaces describe the ports and interfaces which serve as connection points between all components, as described in section 2.3.2. For this, AUTOSAR defines components’ required and provided ports, Sender/Receiver-Interfaces, and Client/Server-Interfaces (see figure 4.1) to indicate different communication patterns. Furthermore AUTOSAR specifies data types like boolean, integer, or float as well as system signals which are exchanged over hardware via BUS systems. Section 4.1.3 will give a more detailed example for the completeness and clearness of AUTOSAR and for the usage of syntax elements.

This rough set of AUTOSAR syntax elements, which are more detailed by the AUTOSAR specification see [10], should show that AUTOSAR provides the possibility to specify the most important elements of an automotive system on the technical level. In the scope of this thesis there can no statement be given about the underlying syntax details of each element. It would rise above the ability of one single person to understand any detail on software, hardware or system side. Therefore it must be trusted in the expertises of several domain experts which are developing the AUTOSAR standard, that AUTOSAR provides a complete or sufficient set of syntax elements.

- **Clearness on meta model level:** Even if all syntax elements are defined completely, it is important for developers to understand the meaning of syntax elements and the context within they are allowed to be used. For this reason the AUTOSAR standard provides a lot of documents, which describe the AUTOSAR meta model in detail. However, there are four most important documents to detail the AUTOSAR meta model: “Software Component Template”[15], “Specification for the ECU Resource Template” [19], “Specification of the System Template” [25], and “Specification of BSW Module Description Template” [20]. Each document details the meaning of syntax elements of the respective template by the use of native text, in addition to the graphical representation of the meta model. Both the documents and the meta model are available to describe the semantics or meaning of AUTOSARs’ abstract syntax and the relationships among its elements clearly. AUTOSAR documents do not only refer to specifications but also to requirements, which must be hold for using AUTOSAR. Consequently,

clearness of the AUTOSAR meta model is ensured by detailed specification documents and requirements, which explain the syntax and its usage.

The detailed description of syntax elements and their semantics on meta level, enables developers to specify concrete automotive systems on technical level by the use of these elements. This leads to the question if concrete specifications as model instances of the meta model are complete and clear.

On model level

- **Completeness on model level:** As it is important that the specification on the model level is complete too, AUTOSAR specifies a methodology (see [13]). This methodology, described in section 2.1.3, specifies all relevant artifacts or work products and dependencies among them. Furthermore, procedures describe relevant artifacts and functions, which are necessary to come from one set of artifacts to another set of artifacts. The procedures provide continuity and traceability between artifacts as well. However, even if the methodology prescribes or recommends artifacts and a chronological order between them, the methodology just has a supporting character. As AUTOSARs' methodology does not intend to prescribe anything, developers themselves must ensure that the content of artifacts is complete. But thereby they are supported by tools.

Some artifacts may contain implemented code and other artifacts contain specification modeled by the use of AUTOSAR. While syntax and individual semantical aspects of code can be verified by the use of compilers, AUTOSAR specified artifacts can be verified by the use of model checkers/analyzers [31, 29], which are able to proof models whether only AUTOSAR syntax elements are used properly and whether constraints, which were specified for these elements on meta level, are hold. Because the AUTOSAR meta model specifies relationships yet, it is possible to proof whether syntax elements on model level possess all predefined relationships to other elements.

A model analyzer, for example, can verify that a ClientServerInterface possesses minimum one OperationPrototype, like described by the AUTOSAR meta model see figure 4.1.

- **Clearness on model level:** Model checkers/analyzers indeed are able to verify a small set of semantics defined by constraints within the meta model, but they are not able to under-

stand whether the specified elements makes sense too. An operation, for example, which is specified requiring four input parameters will be correct for a model checker though the implemented operation requires two input parameters. Such a syntactic inconsistency can just be recognized during integration of components on model or code level, or by the developer itself. In the same way it is not possible to ensure that an implemented operation does the job, which was specified. An operation called `getTemperature()`, which has to return the engine temperature, can also return the environment temperature. This represents a semantical inconsistency between components and currently AUTOSAR solves this semantics problem by the use of textual comments. But because of the problems which come along with textual documents like described in section 3.1.2, textual documents will lead to several interpretations and misunderstandings too. Therefore it is an drawback of the current AUTOSAR standard.

A further drawback concerns the missing abstraction levels of AUTOSAR. As mentioned above, AUTOSAR specifies an automotive system just on a technical level. Higher levels of abstractions, which abstract from technical details are missing. Such abstraction levels may concern the observable behavior of vehicle, a vehicles' environment, vehicles' variants, or (non-)functional requirements. Indeed, figure 4.3 shows that AUTOSAR allows the description of processing unit, resource consumption, or compiler properties. However, this does not suffice to describe all (non-)functional requirements. Section 4.1.4 will show that EAST-ADL2 will extend AUTOSAR by such higher abstraction levels and most of these requirements.

All in all the last section shows that AUTOSAR ensures completeness of specifications as far as possible. By the means of the clear specified meta model, it is possible to avoid misunderstandings (compare to reason number 3 in section 2.4.1), because all available syntax elements posses an unambiguous meaning. But also missing information (compare to reason number 2 in section 2.4.1) can be avoided by prescribed obligatory relationships within the meta model. Individual drawbacks concerning some missing but necessary abstraction levels may be avoided by the use of the EAST-ADL2, as we will see in section 4.1.4. Other drawbacks, which concern the semantics of information and which come along with the use of textual description, stay unclear.

Consistency

In order to ensure the consistency between software components on the code level, it is important that already the software components' specifications are consistent. Like above, consistency con-

cerning the specification must also be hold on different levels or between different things. First of all the artifacts themselves must be consistent. This is discussed section “Consistency between artifacts” below. If multiple artifacts are consistent within one tool, it must also be possible to transfer artifacts to other tools to ensure the interoperability with other developers, companies, or devisions, which mostly do not use the same tool. This is described in section “Consistency between tools”. Finally, if consistent artifacts can be exchanged between tools, consistency must be ensured between different working copies and versions of artifacts. Before this topic will be detailed in section “Maintenance of artifacts”, section “Consistency between devisions and departments” briefly introduces it.

Consistency between artifacts:

- **Consistency on meta model level:** In order to work with AUTOSAR, the large set of documents and the meta model (see [10]) must be hold consistent to avoid inconsistencies among different documents. However, as mentioned above, it is almost impossible to proof or to verify the consistency of all AUTOSAR standard specifications. This does not only concern the textual documents, but also the consistency between meta model and documents, which detail the meta model. Two elements are provided by most documents to simplify the orientation between documents and to understand the life cycle of them: a document change history and a listing of dependent documents to trace between different documents and information. This, on the one hand should restrict consistency checks to dependent documents. On the other hand the change history anticipates question in terms of consistency with earlier versions of documents. Unfortunately there are no further possibilities to proof the consistency, so that consistency again must be based on experiences.
- **Consistency between meta model level and model level:** Consistency between meta model level (this concerns documents and meta model) and model level can be ensured by two things: First of all the AUTOSAR meta model or profile has to be used on the model level for modeling. This restricts general modeling elements, which may be provided by a tool, to AUTOSAR conform syntax. For modeling AUTOSAR specifies so called Modeling Rules (see [26]), from which, for example, one demands that a model shall be compliant to the meta model. And, like above mentioned, a model checker/analyzer [31, 29] can be used to verify that constraints on meta model elements are kept on the model level. This should ensure the consistent usage of meta model elements within concrete instances on model level. Secondly, a developer should also stick to the AUTOSAR standard, which means that developers should use AUTOSAR syntax elements in the right context, that models make “sense”.

- **Consistency on model level:** By then, it can be assumed that everything on meta model level and between meta model level and model level is consistent. After that, the last challenge is to hold models or specifications on model level consistent for enabling the integration of multiple models (compare to model integration 2.3.3). As models should all be designed by the use of the same (AUTOSAR) meta model, it can be assumed that the used syntax of multiple models is equal and consistent. The functionality of AUTOSAR Tools then can be used to ensure the consistency between models, which concerns dependent data and relationships as well. “AUTOSAR Tools are software tools that may occur within the AUTOSAR methodology and support interpreting, processing and/or creating of AUTOSAR models.”[21] Each AUTOSAR tool should keep multiple models consistent within itself. For this, such a tool should provide the same functionality like conventional development environments: syntax checks, updating of changes within all dependent models and packages, and simple semantical checks. Thereby simple semantical checks, for example, concern used data types or matching interfaces. But in order to hold artifacts more consistently, AUTOSAR should introduce some modeling guide lines which ensure consistent naming, structuring, and modeling between multiple developers.

In conclusion it can be assumed that AUTOSAR supports consistency between models or specifications of the same version within one tool.

Consistency between tools:

- **Exchange Format:** As specifications or models have to be exchanged between developers and companies, from which each perhaps uses another tool, it must be possible to exchange models between tools and developers. For this, AUTOSAR specifies a standardized exchange format in form of XML. So called “Model Persistence Rules [14] specify the proceeding to transform concrete AUTOSAR models into a concrete XML description. For this reason the rules also describe how a XML Schema can be generated out from the AUTOSAR meta model in order to validate the structure of a concrete model in XML notation. The persistence rules specify the order of XML tags, the linkage of XML or AUTOSAR components, and the compliance between multiple XML Schemes or documents. Figure 4.4 shows the correlation between AUTOSAR, XML, and Model Persistence Rules. As instance of UML2.0 and applying the AUTOSAR Template Profile, “AUTOSAR Templates” represent the meta model as described in section 2.1.2. These templates can be transformed into a XML Schema representation by using the “Model persistence Rules”. As instance of the AUTOSAR meta model, a concrete model can be specified on model level, which again can be transformed into an

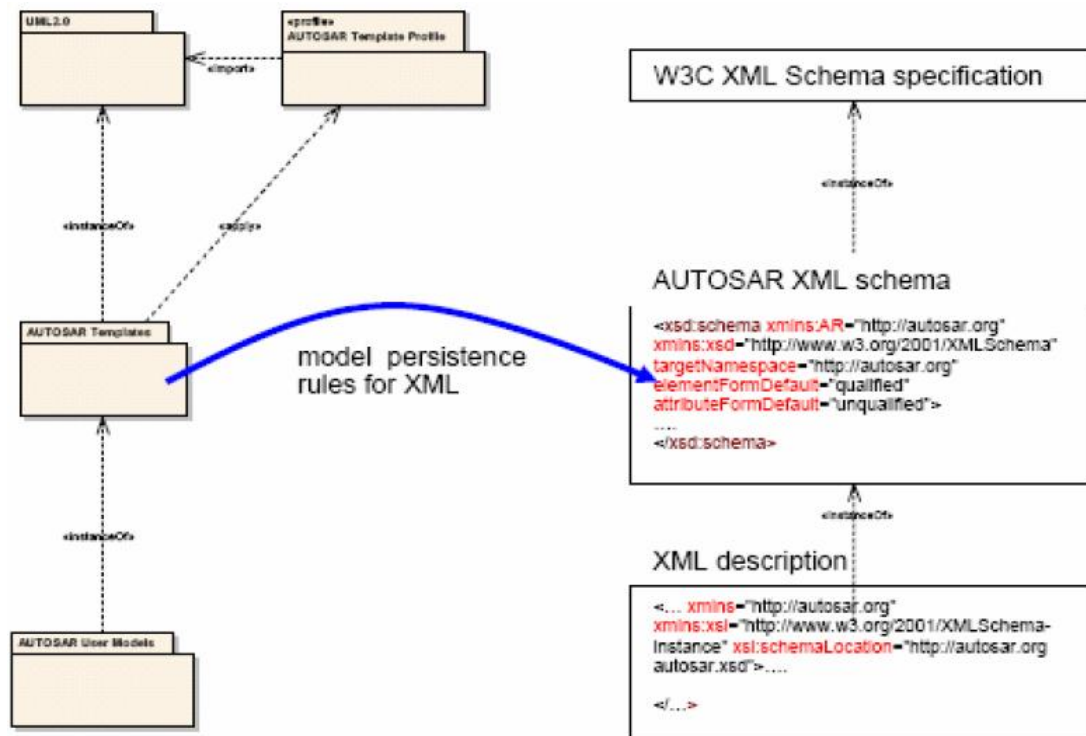


Figure 4.4.: AUTOSAR Model Persistence Rules for XML , from [14]

XML description of these models by using the “Model Persistence Rules” as well. The XML descriptions can then be checked and validated against the XML Schema. As XML is an over-all accepted standard, it should be supported by most of the tools available.

- **Tool interoperability:** Beside this XML based exchange format, in the “Specification of Interoperability of Authoring Tools” [22] AUTOSAR also specifies the interoperability of AUTOSAR tools, which (like mentioned above) support modeling with AUTOSAR, itself. The specification provides some requirements, which should be taken into account by any AUTOSAR tool. The requirements prescribe, amongst others the support for concurrent modeling (e.g. between OEM and supplier and back or between developers), the shipment of AUTOSAR models and related artifacts (e.g. from one tool to an other), the migration between different AUTOSAR and model versions, and the compliance of meta models and models.

Furthermore, AUTOSAR specifies interaction between AUTOSAR and other Behavioral model tools, like statemate or simulink, within the “Specification of Interaction with Behavioral Model” document, see [21]. The document concerns the mapping of AUTOSAR specific models to other functional behavior models tools, which base on more mathematical models than AU-

TOSAR.

These documents and a large set of other specifications and requirements, which would blow the scope of this thesis, enable interaction of tools and model exchange. Holding all these requirements and demands can ensure the interoperability of several tools well. However, even if interoperability of different tools is ensured principally, two drawbacks still remain:

Different vendors try to give models a personal touch and there are different file structures applied by different developers to store multiple models. The personal touch of vendors tries to commit developers to one vendor. For this reason some vendors use differing header files within XML exchange files or they allow additional adapted modeling primitives. These additional modeling primitives may be necessary in the eyes of individual vendors, because AUTOSAR makes no, or only just a few guide lines concerning individual elements deliberately. However, this leads to a violation of the AUTOSAR standard, which may cause a lot of manual reworking and complicates the migration from one vendor to an other. For this reason, each vendor should keep with the AUTOSAR specification without specific adaptation.

In conclusion, the following recommendations should be hold to ensure interoperability of tools:

- AUTOSAR tools must keep with AUTOSAR specifications
- Tool vendors should reduce personal touches by a minimum
- Additional modeling elements, which are not specified by the AUTOSAR standard, must be avoided

Concerning differing file structures and versions, paragraph “Maintenance of artifacts” [4.1.2](#) will give more information.

Consistency between divisions:

- AUTOSAR provides a small but sufficient set of elements to support consistency of models between divisions. The package which aggregates the elements is called “AdminData”, see figure [4.8](#). The “AdminData” can be found in the “GenericStructure” package of the AUTOSAR meta model, which enables almost any other package to use these “AdminData” elements.

“AdminData” aggregate some document revision elements which keep revision information depending on date. Revision information aggregate two further elements: Company specific

revision information (“CompanyRevisionInfo”) and document specific modifications (“Modification”). The “CompanyRevisionInfo” elements are used to generate information on document version within the respective company. In particular it supports the annotation of team members, which provides contact information to other developers. On the other side “Modification” elements are used to record what has changed in one document, in comparison to its predecessor by the usage of textual descriptions.

All in all “AdminData” support consistency between multiple divisions, by annotating an AUTOSAR element with revision versions, company specific information like contact information, and modifications in comparison to other versions of an element. “AdminData” indeed support maintenance of artifacts and their consistency but they do not solve inconsistencies between AUTOSAR models of several departments. Annotated modifications give information about modifications within one model (element), but this is just an indication for developers. Modifications mostly have a stake in other dependent models, which can not be adapted for consistency automatically. This comes along with textual characters of modification descriptions, which can not be interpreted by tools. Therefore, developers have to take these changes into account manually or AUTOSAR tools must be able to detect such changes like demanded by the AUTOSAR standard and mentioned within the last paragraph.

Further aspects concerning consistent file structures or versioning are described in section [4.1.2](#), Maintenance of artifacts.

In conclusion it can be said that AUTOSARs’ consistency support enhances the maintenance of artifacts between tools and divisions (compare to reason number 1 in section [2.4.1](#)). Under the assumptions, that all AUTOSAR models can be hold consistent, AUTOSAR also prevents misunderstandings or wrong/inconsistent information (compare to reason number 2 and 3 in section [2.4.1](#)).

Continuity & Traceability

Continuity between models and model elements should enable the traceability of decisions, understanding of dependencies, and availability of information. This means that it should be possible to define continuous relationships which represent the dependencies between AUTOSAR specifications and/or models.

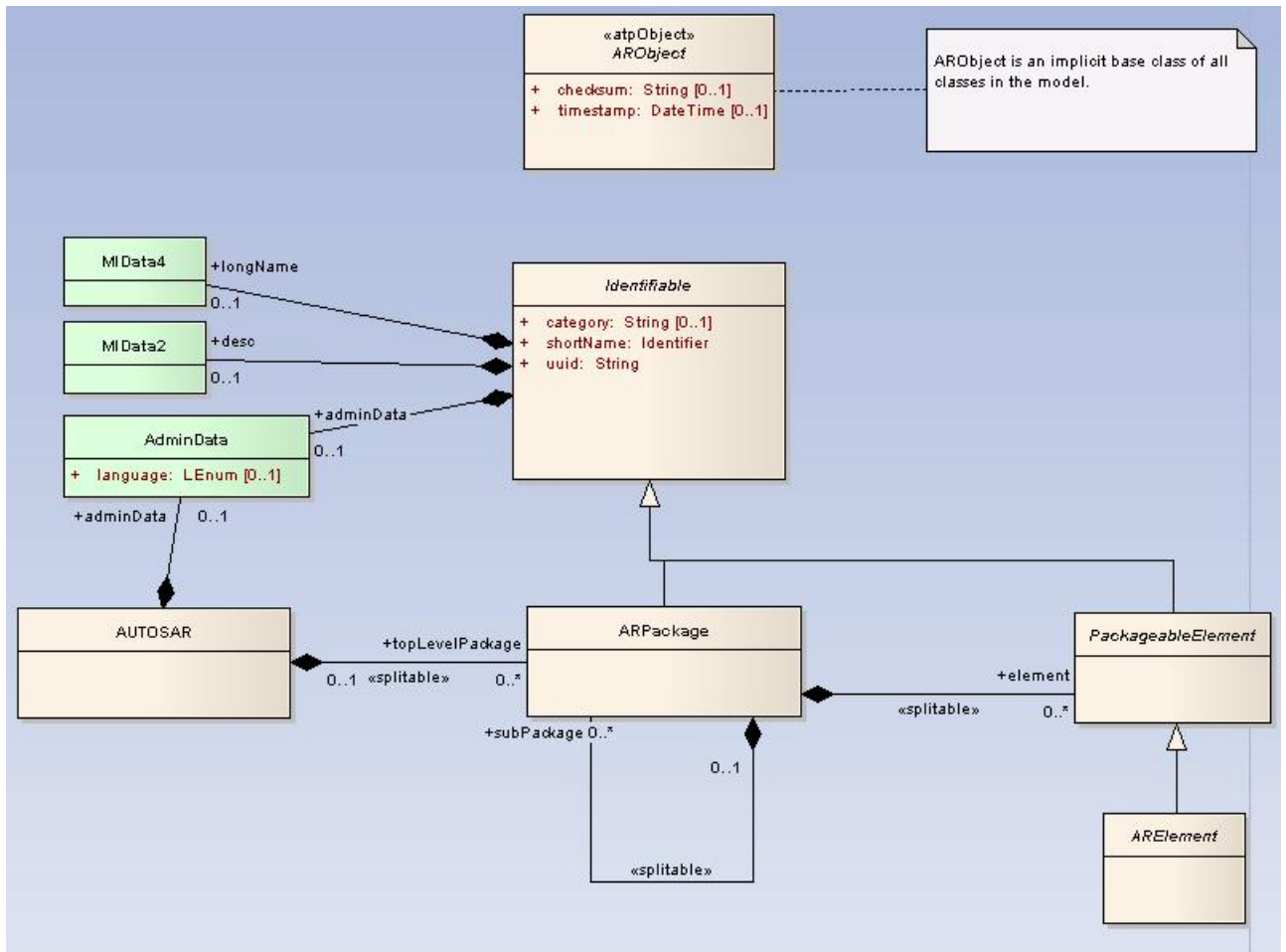


Figure 4.5.: AUTOSAR Top level structure, from [12]

As depicted in figure 4.5 an “AUTOSAR” element aggregates multiple “ARPackages”, which can be subdivided into interconnected sub-packages. Each package again aggregates multiple “PackageableElements”, the superclass of any “ARElement”, compared to figure 4.2. A “ComponentType”, which is derived from “ARElement”, may be an “AtomicSoftwareComponentType”, a “CalprmComponentType” or a hierarchical composition of them, like depicted in figure 4.6. “AtomicSoftwareComponentTypes” can thereby be subdivided into application software, basic services, drivers, ECU abstraction software, or sensor/actuator software. The characteristic of all of these elements is, that they are all interconnected and aggregated into one root element, called “AUTOSAR”. This integration of multiple different models (compare to model integration 2.3.3) is applicable regardless whether the elements specify software, hardware, the entire system, or a mixture of them. Thus it is possible for example to define an application software within one department, while a second department specifies a basic software description. After completion the two elements can be interconnected and

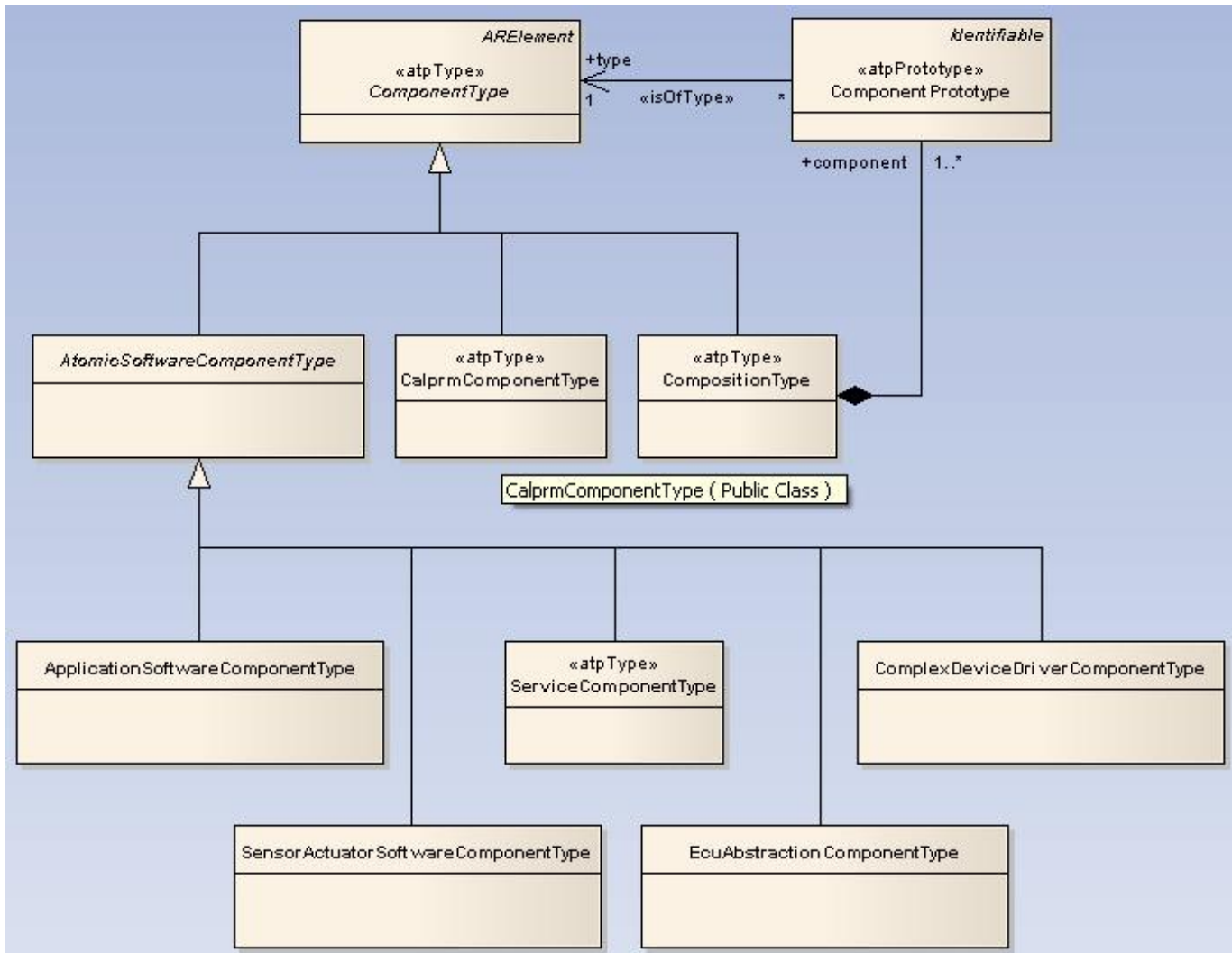


Figure 4.6.: AUTOSAR components and composition, from [12]

aggregated into one “AUTOSAR” element simply.

So, an “AUTOSAR” element provides availability of all specifications concerning the systems’ hardware and software components inside of one model. As all elements may be interconnected via relationships, not only the availability (compare to reason number 1 in section 2.4.1) of information is ensured, but also dependencies among elements can be defined clearly in order to enhance the understanding of correlations (compare to reason number 3 in section 2.4.1).

Furthermore, AUTOSAR indeed allows tracing between sub-packages and model elements, but AUTOSAR does not provide traceability of modeled solutions. As AUTOSAR just focuses on technical specifications, AUTOSAR does not contain specifications or artifacts concerning non-technical infor-

mation (e.g. environment modeling or modeling which concerns perceptible vehicle functions). This leads to technical design solutions modeled with AUTOSAR exclusively, whereas it will not be clear why exactly this solution was chosen. Thus it is an AUTOSAR drawback, which hampers a continuous development process, that AUTOSAR does not support continuity between all process relevant artifacts or stakeholders.

Although section 4.1.4 will show that EAST-ADL2 complements AUTOSAR with such artifacts, AUTOSAR can also be used in the context of other frameworks or ADLs which provide some missing artifacts. However, while EAST-ADL2 is aligned with AUTOSAR, other ADLs are not. Due to the differing syntax or semantics, inconsistencies or mismatching between AUTOSAR artifacts and artifacts provided by other frameworks may be caused. This missing continuity again causes manual reworking and a lot of adaptations in order to integrate AUTOSAR artifacts with other ones.

Integration of all disciplines

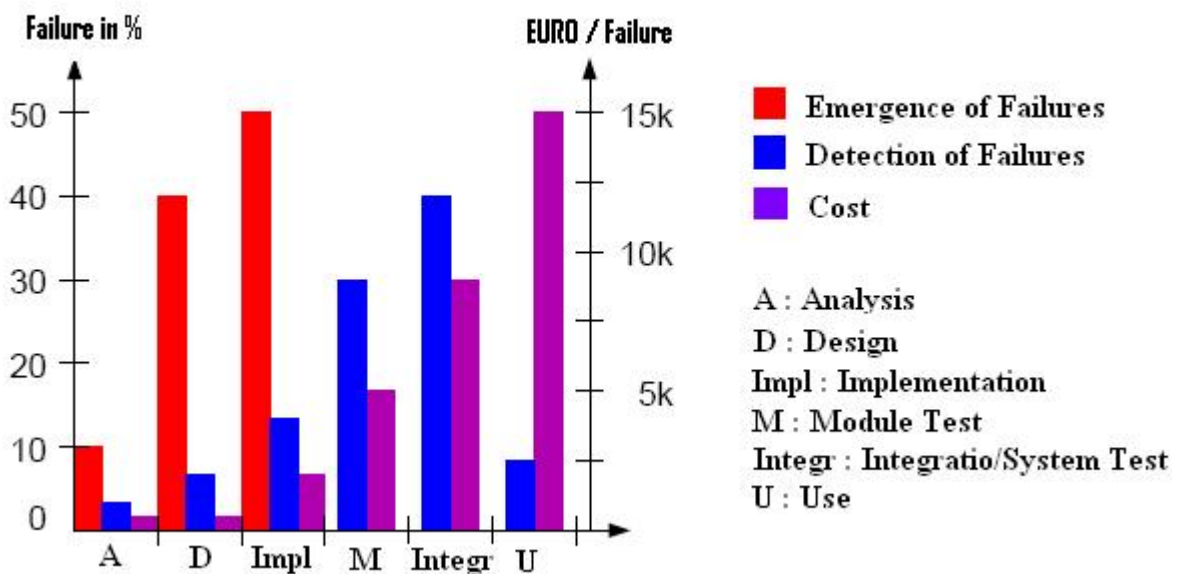


Figure 4.7.: Emergence and Detection of Failures vs. Costs, from [54]

The aforementioned continuity also concerns the integration of different disciplines and stakeholders into development. Multiple development disciplines are important because each discipline has its own specialized view on systems' needs, which can not be covered by a single instance or expert. Amongst others, such disciplines may be requirements and variants engineering, software engineer-

ing, hardware engineering, systems engineering, integration, or validation & verification.

Like mentioned above and depicted in figure 2.3, AUTOSAR covers the system, the ECU (hardware), and software. For this reason one can assume that AUTOSAR supports all activities concerning the systems engineer, the hardware engineer, and the software engineer. However, AUTOSAR only focuses on a technical side of development close to implementation. Because no one designs a software architecture from scratch on implementation level, it is also necessary to support disciplines from the level before implementation, like requirements, analysis, or (non-)functional design. AUTOSAR does not provide support on these levels for system, software, or hardware engineers. Furthermore, AUTOSAR also does not support validation & verification. This means a critical drawback for AUTOSAR, because a missing support for individual disciplines requires the usage of means which are not conform with AUTOSAR. Section 4.1.4 again will show that EAST-ADL2 complements AUTOSAR with missing disciplines. This again is only possible because EAST-ADL2 is aligned with AUTOSAR, its syntax, and its semantics.

On the other side AUTOSAR indeed does not support the discipline of integration explicitly, but implicitly. By the fact that AUTOSAR models can be integrated into an entire model, which can be validated, checked and simulated as mentioned above, an integrator is also able to integrate components on an earlier stage of development before implementation. This kind of integration conforms to model integration (virtual integration) like described in section 2.3.3. Because models are available much earlier than the code, integrators are able to find out integration drawbacks, which would avoid integration on implementation level, already on model or specification level. This possibility to integrate models already before the executable code avoids integration problems, which cause more time and money on implementation level than on the early model level. This is also accounted with some studies which confirm, that faults which are recognized on specification level cause fewer costs or reworking than on implementation level or later. Figure 4.7, for example, depicts an actual situation of development and phase specific error-proneness in comparison with error detection and costs caused by failures. The figure shows that failures are made inside of early development phases, but tardily detected and fixed expensively. This means that the earlier a failure can be detected by an integrator, the fewer effort for an integrator emerges from fixing these failures.

In conclusion it can be said, that AUTOSAR integrates all disciplines concerning system, software and hardware on technical level. Furthermore, AUTOSAR supports, even if not explicitly, the activities of an integrator by providing a detailed system overview. Other disciplines are neglected by

AUTOSAR caused by its technical focus. However, they should be supported by other frameworks, which can be linked with AUTOSAR.

Maintenance of artifacts

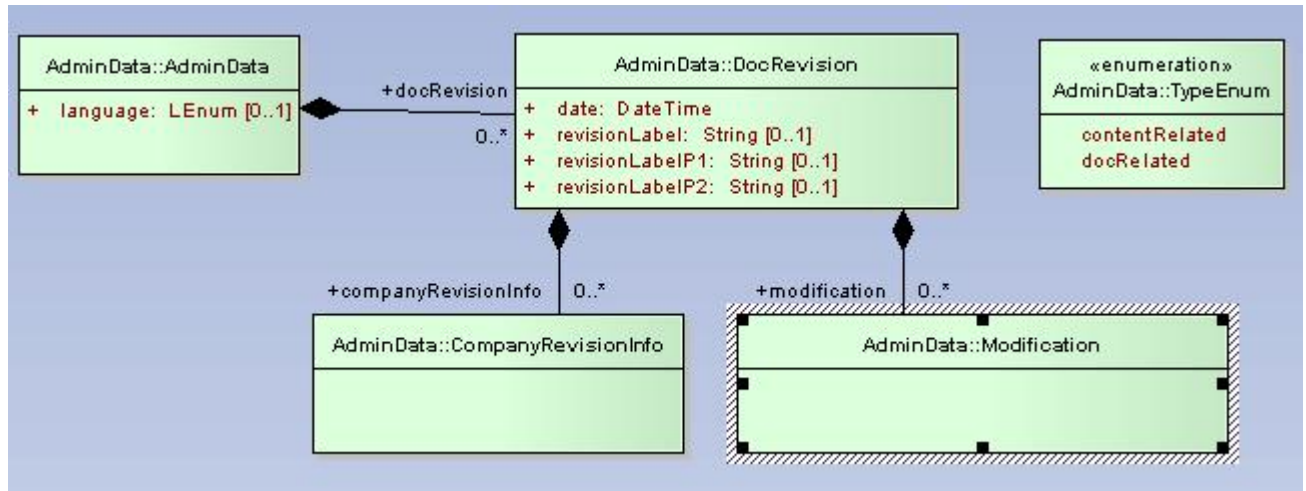


Figure 4.8.: AUTOSAR GenericStructure:CommonPatterns:AdminData, from [12]

Folder name	Content
src	Source files (*.c, *.asm)
inc	Header files (*.h)
lib	Library files (*.lib)
doc	Documentation files (*.doc, *.pdf, *.txt, ...)
mak (module related)	Makefiles (*.mak)
mak (test case related)	Makefiles (*.mak)
<compiler >	MemMap.h & xxx_compiler_config.h (*)
obj	Object code files (*.o, *.obj)
gce_input	configuration parameter definition files (Input for Generic configuration editor)
gce_output	Output for generic configuration editor
generator	The module specific generator (tool)

Figure 4.9.: Example AUTOSAR file structure, from [17]

Section “Consistency between deviations” has described the possibility of AUTOSAR to annotate models and model elements with revision information, which contain version and modification information of AUTOSAR model elements. These information should keep consistency among different versions of one artifact or model. But because of the large set of information and specifications of

an entire vehicle, it is not practical and almost not possible to hold all information within one unique model. For this reason the set of artifacts is split into multiple files, folders, and sub-folders, which again are subdivided into different versions and configurations.

In order to ensure availability of documents it must be possible to orient oneself within the resulting large set of files and folders. Especially concerning the exchange of configuration packages it is important that file structures match between multiple divisions. Because if an exchanged configuration of a division A does not match the structure of a division B, it takes a lot of time to adapt files for the intra-divisional file structures. Furthermore, changes and updates on files, artifacts, and configurations must be communicated well, in order to ensure that nobody works with obsolete or wrong working copies.

The AUTOSAR standard specification makes no guide lines concerning file structures or configuration changes. The “Specification of Interoperability of Authoring Tools” [22] of AUTOSAR indeed recommends that AUTOSAR tools shall support sets of files for example by creating and interpreting meta data, but there are no concrete recommendations given inside of that or another specification.

That finding also was brought by some AUTOSAR test projects. The projects found out, that the storage structure on delivery folder or a SVN (SVN stands for Subversion and represents a version management system) should be compatible to the working structure for each module [27]. This means that a file structure must be located by using the same path on all work stations. For this reason, “File Structure of Integrator 2”[17] document, which has not been published so far, specifies an intra-divisional file structure to simplify the file management and artifact integration. This document prescribes several directory levels, naming rules of files and directories, and the content of respective directories.

For example: The naming rules specify that the length of the directory name should be as short as possible. Allowed characters are small letters, numbers, and underlines. Moreover, the name of the main directory shall be a concatenation of module name, vendor name, and hardware platform. Beside naming rules figure 4.9 depicts obligatory subdirectories of two second level directories “ssc”, which stands for “standard software core”, and “cfgY”, where cfg stands for configuration and Y is denoting the specific number of the configuration. The directories shall have the specified content. Unfortunately, such detailed information about file structures are not given within the AUTOSAR standard specification. However, in order to avoid irritating reworking and to support availability of artifacts, the following rules for consistent file structures, which base on [17], should be adopted by

the AUTOSAR standard:

- Folder structure rules should be prescribed, which define the minimal/maximal depth of directory levels and obligatory folders on each level
- Folder content rules should be prescribed, which define the kinds of files or artifacts, which are allowed or obligatory inside of an individual folder
- Naming rules should be prescribed, which prescribe the building of names, and allowed literals

Even if the file structure of one configuration or a set of files will be clear, it is not ensured that also a right or actual version of a file or artifact is used. An AUTOSAR model indeed allows an annotation of such version information by the use of “AdminData”, but this is just local solution for one file or artifact. Concerning the evolution of configurations and files within and between different baselines and releases, it is important to manage changes and influences of changes on dependent artifacts globally among different divisions. It is not possible for divisions to check each single file or model of a configuration for modifications or their dependencies to other files. Therefore not only the management of changes and versions is important, but also the communication of them to other developers. In this context section 2.3.4 mentioned two tools: Telelogic Change and Telelogic Synergy. While “Synergy” manages configurations, “Change” is responsible for change management and communication. However, both tools are not part of the AUTOSAR specification. Therefore it is not ensured that only these tools are used or that they are used in the same manner when different divisions are working together. In this regard the BSW module integration project from section 2.3.4 tries to define a cross-divisional process, which should support these questions. However, although the experiences from that project must be awaited to make further assumptions, the following basic recommendations based on the survey should be hold and adopted by AUTOSAR perhaps:

- There should be a central/global repository or “SharePoint” for configurations and changes
- Each developer must be able to use the most actual, but released artifact
- Changes must be communicated to all involved parties as quick as possible
- It must be communicated what is new compared to other baselines and builds
- It must be communicated what has changed compared to baselines and builds
- It must be communicated which dependencies to other modules or baselines are concerned by a change
- It must be communicated which fault was resolved by a change

- It must be communicated who made a change

Assignment of roles and responsibilities

AUTOSAR and its Methodology, see [13], give no guide lines concerning roles and responsibilities, like described in section 2.1.3. AUTOSAR sets roles and responsibilities aside in order to apply AUTOSAR within any development process, which differs from division to division, individually.

However, it is important to define roles and responsibilities in the scope of one or more development processes. Because roles are coupled with disciplines, it makes no sense that each divisions assigns the roles individually. In that case it could happen that experts of different disciplines have to communicate about the same artifact during a cross-division development. Due to different technical backgrounds and knowledge base of different disciplines (compare to section 3.1.2), this comes along with misunderstandings and faults. Furthermore, if roles and responsibilities are not assigned before the development, it could happen, that nobody is responsible for individual artifacts. This could lead to forgotten artifacts and to extensive reworking ex post.

So it is a benefit of AUTOSAR that it is integratable into any development process and division. On the other side, missing role assignments also represent a drawback of AUTOSAR, which leads to misunderstandings and missing artifacts. For this reason it would be important for AUTOSAR to provide a framework of roles and responsibilities, which can/should be applied by divisions. But because the BSW module integration process, which is presently under development like described in section 2.3.4, tries to define roles more explicitly, the experiences of that process should be awaited before further recommendations are made concerning roles and responsibilities.

Conclusion AUTOSAR Support to enhance Basic Conditions

In conclusion, figure 4.10 summarizes all results concerning basic conditions of development, which can be enhanced by AUTOSAR. Thereby the arrows indicate how far AUTOSAR provides support for a current critical point. Unfortunately, there is no measurement, which measures enhancements exactly. For this reason the grade of enhancement was assessed subjectively on the basis on variety and the helpful character which is provided by AUTOSARs' specifications.

- **Completeness & Clearness** (of Standards and Descriptions)

- **Consistency** (between Artifacts, Tools, and Divisions)

- **Continuity & Traceability**

- **Integration of all disciplines**

- **Maintenance of artifacts**

- **Assignment of roles and responsibilities**


Figure 4.10.: AUTOSAR Support to enhance the basic conditions

One can see, that AUTOSAR supports completeness, clearness, and consistency for artifacts very well, while roles and responsibilities are not supported at all. Therefore it is very important to find solutions, which support this concern. Inbetween there are critical points concerning the continuity between artifacts, different disciplines, and maintenance of artifacts, which are supported by AUTOSAR. However, in comparison with the green arrows within figure 4.10, these points should be extended in order to simplify integration further more.

Therefore, it would be advisable and a first step for enhancing the basic conditions, if guide lines or best practices would be established more than today. Such rules should concern amongst others the aforementioned six points in order to ensure, that any party handles the basic conditions similarly to other involved parties. This simple step could help to avoid inconsistencies or irritations yet.

4.1.3. Concrete Inconsistencies & AUTOSAR

Beside reasons for inconsistencies, there are also some concrete inconsistencies (compare to section 2.4.2), which may be avoided by using existing AUTOSAR concepts too. If the AUTOSAR standard would be applied by any OEM or supplier, the common base could avoid most of the problems, faults, or failures. On the basis of the points, which were identified in section 3.2.2, the following will show how far AUTOSAR is able to avoid some concrete inconsistencies, which were criticized within the survey.

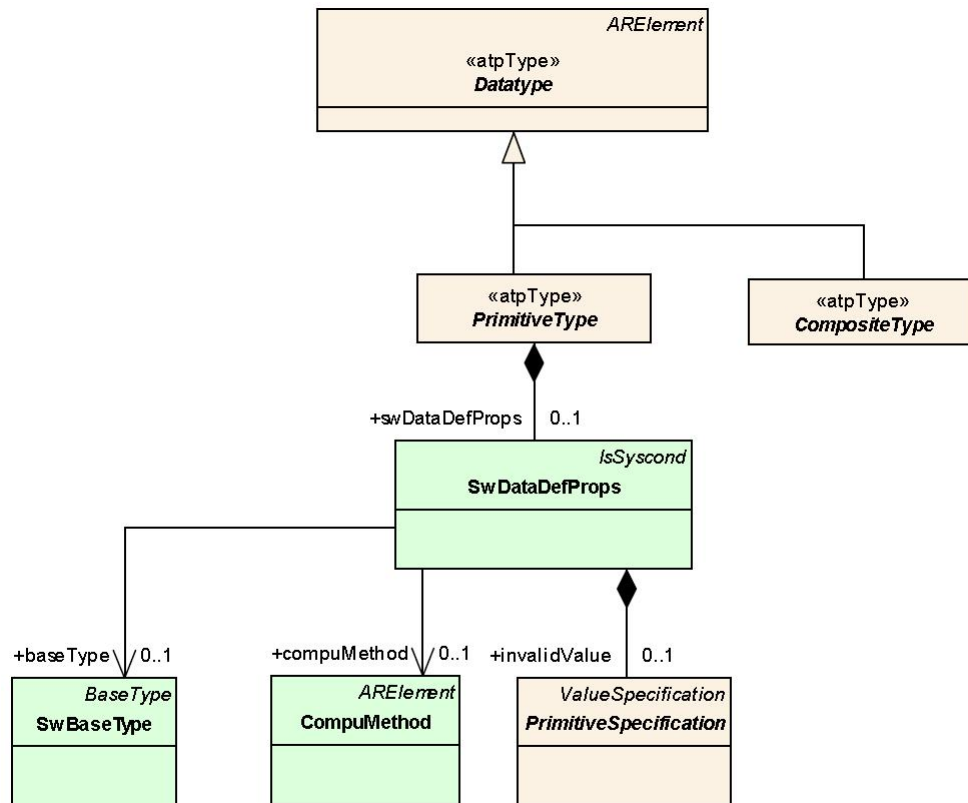


Figure 4.11.: AUTOSAR Data Types Overview, from [12]

Static semantics of interfaces

The survey has shown that interfaces, by which components are interconnected and integrated, do not match sometimes. This can be caused by static or dynamic mismatches between two interfaces. While static interface properties have an impact on the static connection of components, the dynamic properties have an impact on the behavior between two components at runtime. Both properties must be specified and implemented properly to enable the interaction of components. For this reason, the following section analyzes AUTOSARs' possibilities to describe the syntax or static properties of interfaces in a standardized manner. Subsequently section "Dynamic semantics of interfaces" concentrates on the dynamic properties of interfaces.

AUTOSAR language elements to describe static semantics of Ports & Interfaces:

Generally AUTOSAR specifies ports and interfaces to describe the static semantics of interaction points. A component can define some ports, which are associated with an interface, like depicted in figure 4.1. A port can be a required port or a provided port. While a required port indicates, what kind

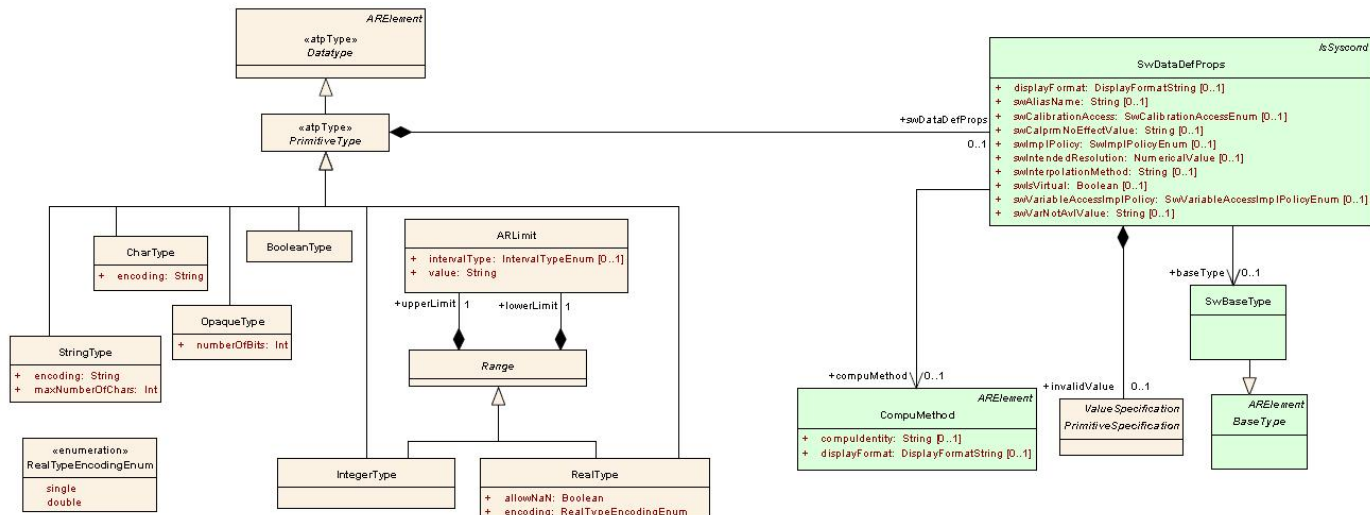


Figure 4.12.: AUTOSAR Primitive Data Type, from [12]

of services are required by a component, the provided port indicates, what services a component provides to other components. “In addition to the formal specification required to implement the communication via ports, a “PortPrototype” can carry so-called Port Annotations. They do not directly influence the signature of calls via this port, but contain further information useful for the application developers of the components on both sides of the connection. [15]” Annotations may be very useful for developers to understand details of ports concerning their usage. However, the drawback here is that annotations are not standardized textual descriptions. This may lead to misunderstanding as well as misinterpretations.

The interfaces themselves can be subdivided into different communication patterns. “SenderReceiver-Interfaces” allow for specification of the typical asynchronous communication pattern where a sender provides typed and named data (compare to “DataElementPrototype” in figure 4.1) that is required by one or more receivers. Due to the unambiguous definition of data, which are sent or received by “SenderReceiverInterfaces”, this kind of communication can be well described on syntactical level.

On the other side the underlying semantics of a “ClientServerInterfaces” is that a client may initiate the execution of an operation, see “OperationPrototype” in figure 4.1, by a server that supports the operation, which is provided by a “ClientServerInterfaces”. The server executes the operation and immediately provides the client with the result (synchronous operation call) or else the client checks for the completion of the operation by itself (asynchronous operation call). Figure 4.1 also shows that “ClientServerInterfaces” aggregate minimum one operation. For each operation it is possible

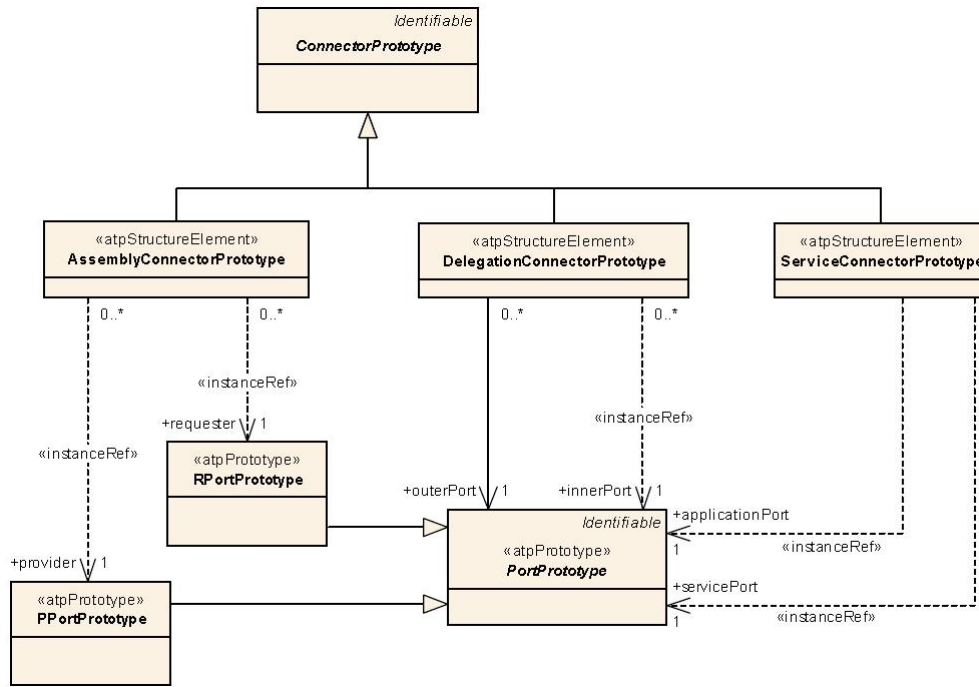


Figure 4.13.: AUTOSAR Connectors, from [12]

to define an ordered set of typed, named, and directed parameters. The direction kind of parameters specifies whether a parameter is input, output or both. Therefore its possible to specify the syntax of all operations inside of “ClientServerInterfaces” exactly. By using AUTOSAR, the name of an operation, the number and order of parameters, the data types of input parameter, and the result parameter can be defined. All properties of a signature are well-described, and the syntax of operations can be described unambiguously, which avoids mismatching operation calls between components.

In addition to that, section 4.1.2 has already described, that AUTOSAR enables the building of compositions of components or “ComponentProtoTypes” typed with “ComponentType”(see figure 4.6). As compositions of “ComponentProtoTypes” have to be interconnected as well, a “CompositionType”, see figure 4.6, also aggregates the abstract meta-class “ConnectorPrototype”(see figure 4.13). A “CompositionType” also exposes PortPrototypes to the outside world. However, its PortPrototypes are only delegated and do not play the same role as “PortPrototypes” attached to “AtomicSoftware-ComponentTypes”. Figure 4.13 depicts the three kinds of “ConnectorPrototypes”, which can be associated with a “CompositionType”:

- “AssemblyConnectorPrototypes to interconnect PortPrototypes of ComponentPrototypes that

are part of the CompositionType as well as”[15]

- “DelegationConnectorPrototypes to connect from “inner” PortPrototypes to delegated “outer” PortPrototypes.”[15]
- “ServiceConnectorPrototype is exclusively used in the context of ECU configuration phase, and must not be used within CompositionTypes of software applications.”[15]

Like depicted in figure 4.11 and figure 4.12, the used data types also are predefined by AUTOSAR. Figure 4.11 shows, that AUTOSAR distinguishes between primitive data types (character, string, opaque, boolean, and real) and complex data types (array and record) in form of “CompositeTypes”. That data types are used for “SenderReceiverInterfaces” as well as for arguments of an operation inside of “ClientServerInterfaces”. By the means of these predefined data types, static type checks are possible to ensure that exchanged data match in terms of their data type (name). Further semantical properties of data types, which also have to match when data are exchanged, are handled in section 4.1.3.

Finally, AUTOSAR also specifies the compatibility of data types, of “DataElementPrototypes” in the scope of “SenderReceiverInterfaces”, of Sender Receiver Interfaces, of Operation Prototypes and their arguments in the scope of “ClientServerInterfaces” as well as the compatibility of “Sender-ReceiverInterfaces” and “ClientServerInterfaces” themselves. See the Software Component Template specification [15] for more details.

Benefits from AUTOSARs static semantics:

The detailed AUTOSAR syntax enables describing all aspects of component interfaces and ports well. At least on the static and syntactical level the connection points can be described in a standardized and complete manner, as described above. That standardized and complete possibility to describe interaction points also avoids some inconsistencies between software components on design level or on implementation level, which must be consistent within the design level (as mentioned above the consistency between design and implementation level can be ensured by the use of code generators). The inconsistencies which can be avoided are described in the following:

- **Syntactical inconsistencies** concerning interfaces, ports, signatures, or data names and types, compare to section 2.4.2 inconsistency 1, can be avoided by the aforementioned unambiguous syntax elements of AUTOSAR. As interfaces can be related already on model level, it is possible to match required and provided ports against each other. On the one side operations, their signatures, and used data types for “ClientServerInterfaces” can be matched.

On the other side it is also possible to match exchanged data (types) of “SenderReceiverInterfaces”.

- **Semantic inconsistencies** can be avoided partly. At least language elements of the AUTOSAR meta model are defined unambiguously, which avoid language inconsistencies, compare to section 2.4.2 inconsistency 2, between used meta model elements. For example, AUTOSAR defines the meaning and the allowed context of operations and parameters clearly.
- **Pragmatical inconsistencies** concerning the communication pattern, see section 2.4.2 inconsistency 1, can be avoided, because patterns are prescribed by the used interface. As interfaces can already be related on model level, it is possible to check whether their used communication patterns match. Furthermore it is possible to link from the software representation to its hardware description provided by the ECU Resource Template by the means of “ComplexDeviceDriverComponentType”, “EcuAbstractionComponentType”, and “SensorActuatorSoftwareComponentType” (see figure 4.6). By the means of that linking it is possible to describe which software component may access a certain hardware.

Table 4.2 summarizes these results. The table shows for each inconsistency (categorized in section 2.4.2), which can be avoided, the respective AUTOSAR meta class, which mainly is responsible for avoiding the inconsistency.

AUTOSAR Interfaces Example(static):

The following example exemplifies the definition and usage of interfaces as well as their interaction in the scope of the AUTOSAR Memory Stack modules. As the memory stack is composed of basic software components exclusively, the example is about Basic Software Integration like described in section 2.3.2.

For this reason figure 4.14 depicts a module overview of the memory hardware abstraction layer inside the basic software layer of AUTOSAR. The figure shows the NVRAM (Non-volatile Random Access Memory) Manager. This NVRAM Manager (NVM) shall provide services to ensure data storage. It shall abstract from the underlying modules and administrate NV data. For this, the NVM can access the “Memory Abstraction Interface”, which abstracts from several memory abstraction modules like an EEPROM and/or a Flash EEPROM emulation device. Such hardware memory abstraction modules again abstract from underlying drivers, which are responsible to access the micro controller hardware. The figure shows that ports are used to interconnect the different modules. The ports provide interfaces to other basic software modules, to application software modules, and to hardware.

	avoidance possible	no special support
Syntactic Inconsistencies		
inconsistent data exchange format		X
inconsistent interfaces	PortInterface	
inconsistent ports	PortProtoType	
inconsistent signatures	OperationPrototype	
inconsistent data names	DataType	
Semantical Inconsistencies		
numerical inconsistencies		X
language inconsistencies	AUTOSAR meta model	
reference system inconsistencies		X
Application-based Inconsistencies		
violation of states		X
violation of obligatory relations		X
restricted data ranges		X
Pragmatic Inconsistencies		
concurrency		X
access restriction on extern resources	ComplexDeviceDriverComponentType, EcuAbstractionComponentType, SensorActuatorSoftwareComponentType	
timing requirements on hardware		X
absolute timing requirements		X
relative timing requirements		X
communication pattern	PortInterface	

Table 4.2.: Inconsistencies avoided by AUTOSARs' static semantics

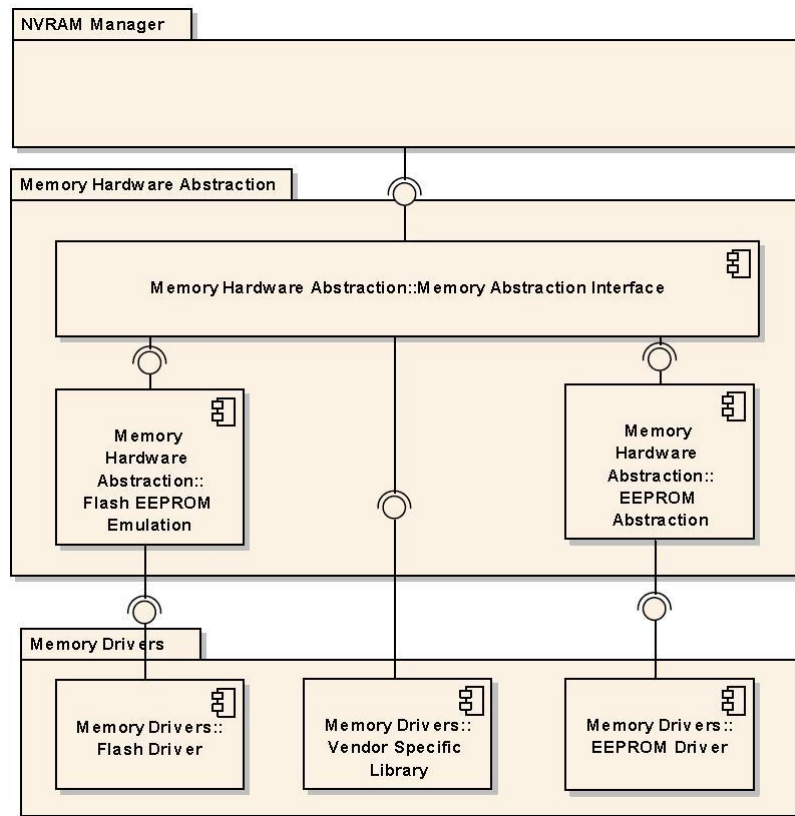


Figure 4.14.: AUTOSAR Memory Management Overview, from [23]

Because of the standardized character of the memory management or other basic services, each basic module can prescribe an API more than application software components. This API can be used either for other basic software modules, application software modules or hardware. The API of a respective module prescribes used data types as well as functions/operations, which are summarized within some predefined interfaces. Figure 4.15 depicts a “ClientServerInterface”, compare to figure 4.1, of the NVM, which summarizes functions specified within the API specification of NVM, see [24]. This “ClientServerInterface” is called *NvMService*, which indicates that it represents an AUTOSAR service of the BSW layer. The API specification details all operations, which is exemplified for the “WriteBlock” operation of the “NvMService” in figure 4.16. One can see that AUTOSAR prescribes the signature (operation name and parameters), reentrancy, and whether an operation can be used synchronous or asynchronous. All information prescribed by an API specification can be modeled by syntax elements the AUTOSAR meta model depicted in figure 4.1 and described above.

As each basic software module prescribes an API in this manner, basic modules can be integrated by using this standardized API. In comparison to figure 4.14, figure 4.17 exemplifies the interaction


```

ClientServerInterface NvMService {
    PossibleErrors {
        E_NOT_OK = 1
    };

    // the next operation is always provided
    GetErrorStatus( OUT RequestResultType RequestResultPtr );

    // the next two operations are always available, but are needed
    // only for "data set" block types. Thus they shall be provided in
    // the interface only for those block types
    SetDataIndex( IN Uint8 DataIndex );
    GetDataIndex( OUT Uint8 DataIndexPtr );

    // this operation is only provided via optional configuration
    // NvmSetRamBlockStatusApi
    SetRamBlockStatus( IN Boolean BlockChanged );

    // the next three operations are only provided for
    // NVRAM API configuration class 2 and 3
    ReadBlock( IN DstPtrType DstPtr, ERR{E_NOT_OK} );
    WriteBlock( IN DstPtrType SrcPtr, ERR{E_NOT_OK} );
    RestoreBlockDefaults( IN DstPtrType DstPtr, ERR{E_NOT_OK} );

    // the next two operations are only provided for
    // NVRAM API configuration class 3
    EraseBlock( ERR{E_NOT_OK} );
    InvalidateNvBlock( ERR{E_NOT_OK} );
};

```

Figure 4.15.: NVRAM Manager Client/Server Interface, from [24]

of memory modules by using the specified interfaces and operations.

For example: The NVM can call the “MemIfWrite” operation specified within the “Memory Abstraction Interface”. The “Memory Abstraction Interface” has to decide whether it uses the “EEPROM Abstraction” module or the “Flash EEPROM Emulation” module to execute the write operation. For using the “Flash EEPROM Emulation” module, the “Memory Abstraction Interface” module can call the “FEE_Write” operation, which calls the “Flash_Write” operation of the “Internal Flash driver” module. Then the driver can write into flash storage.

Application software can use this API too in order to use the services of basic software. To enable integration of application software as well, their interfaces can be described in the same manner, by using AUTOSAR syntax elements of the “Software Component Template”. Therefore it is possible to describe all interfaces between components completely and consistently. The standardized char-

Service name:	NvM_WriteBlock	
Syntax:	<pre>Std_ReturnType NvM_WriteBlock(NvM_BlockIdType BlockId, const uint8* NvM_SrcPtr)</pre>	
Service ID[hex]:	0x07	
Sync/Async:	Asynchronous	
Reentrancy:	Reentrant	
Parameters (in):	BlockId	The block identifier uniquely identifies one NVRAM block descriptor. A NVRAM block descriptor contains all needed information about a single NVRAM block.
	NvM_SrcPtr	Pointer to the RAM data block.
Parameters (inout):	None	
Parameters (out):	None	
Return value:	Std_ReturnType	E_OK: request has been accepted
		E_NOT_OK: request has not been accepted
Description:	Service to copy the data of the RAM block to its corresponding NV block.	

Figure 4.16.: NVRAM Manager API function: NvM_WriteBlock, from [24]

acter of ports and interfaces, makes it easy to integrate other modules or to use basic services by application software.

Dynamic semantics of interfaces

AUTOSAR language elements to describe static semantics of Ports & Interfaces:

All aforementioned elements to describe interfaces, only represent static properties to describe the structure of interfaces.

In order to specify the real dynamic semantics of interfaces as well, AUTOSAR provides a so-called “InternalBehavior” for all “AtomicSoftwareComponentType” and an additional “BswBehavior” for basic software modules, as depicted in figure 4.18. The figure also does show, that each kind of implementation possesses exactly one behavior, which describes the dynamic behavior of an implementation. The depicted tripartition separates implementation from behavior and static component description, which enables to exchange individual component properties.

Figure 4.19 gives an overview about elements, which are provided by AUTOSAR to describe “InternalBehaviors”. The “InternalBehavior” aggregates so-called “RunnnableEntities”(Runnables), which represent the smallest code-fragments that are provided by a component and executed at runtime. Runnables for instance are set up to respond to data reception or operation invocation on a server.

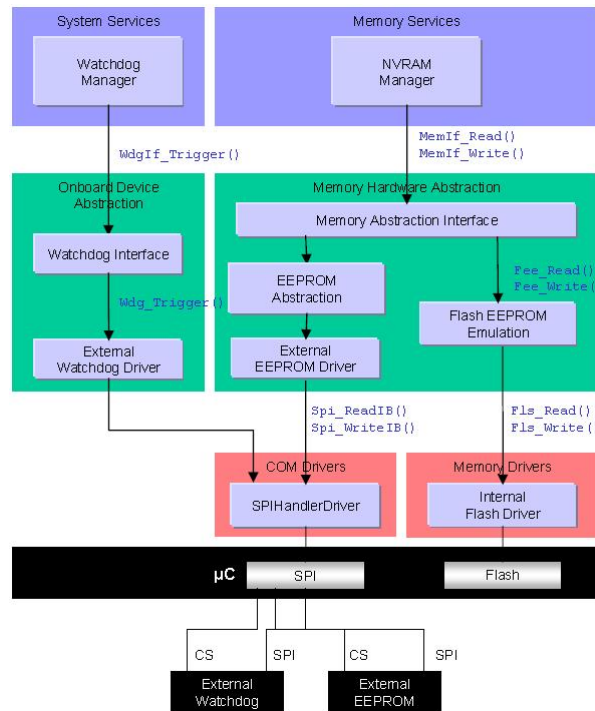


Figure 4.17.: Interface interaction of Layers, example “Memory Management”, from [18]

Therefore they specify which data may be accessed for writing or reading or which data should be sent or received. Moreover, runnable entities may associate “ExclusiveAreas” and specify whether they can be invoked concurrently or not.

“InterRunnableVariables” as well as “ExclusiveAreas”, which prevent an executable entity (runnable) running in the area from being preempted, can be used for communication among runnables. For this reason an “InternalBehavior” aggregates some “ExclusiveAreas”, which can be referred by runnables. In addition, the “InternalBahvior” also aggregates so-called “RTEEvents”, which will occur during the execution. Such “RTEEvents” may indicate, for example, the reception of a remote invocation of an operation on a provided port or a timeout on a required port, which is not receiving the data it expects to receive. However, runnables may also specify some waiting points, which indicate that a runnable waits (a certain time) for a specific “RTEEvent”.

So-called “ServiceNeeds” are aggregated by an “InternalBahvior” in order to provide detailed information about what a software component (application software or basic software) needs from (other) basic software components when it has to be integrated on an actual ECU. By the means of “ServiceNeeds” in combination with information of the BSW Module Template (“BSWImplemen-

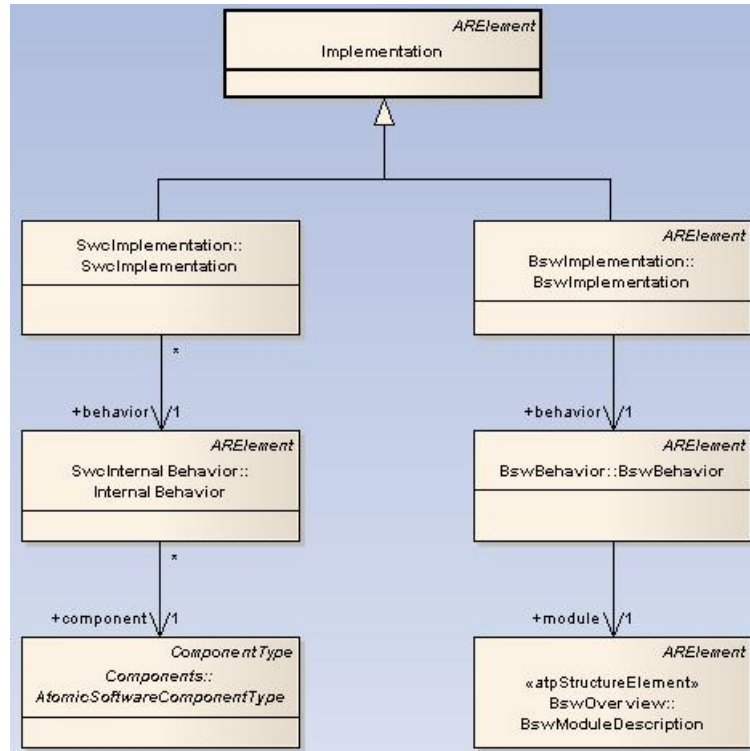


Figure 4.18.: AUTOSAR Implementation of software components and basic software, from [12]

tation” and “BSWModuleDescription”, see figure 4.18), it is possible to generate the “BSWBehavior” automatically. As the “BSWBehavior” is similar to “InternalBehavior” as far as possible (a “BSWBehavior” also uses events, exclusive areas, and runnable entities called “BSWModuleEntity”), only the “InternalBehavior” is described here.

Figure 4.20 exemplifies the usage of “InternalBehavior” elements. One can see, that an “InternalBehavior” aggregates some events, which start runnable entities of the behavior. There are a lot of events (see figure 4.21), which can be defined to execute an individual runnable entity. By the means of events and runnables, it can be specified how an internal behavior or implementation can react to dynamic influences (RTEEvents) from the outside world of an implementation.

As one can see in figure 4.1 a “SenderReceiverInterface” also aggregates a so-called “ModeDeclarationGroupPrototype”, which associates a concrete “ModeDeclarationGroup”. The class “ModeDeclarationGroup” has been introduced to support the grouping of modes and (on model level) to provide predefined sets of modes that could be standardized and re-used. The set of modes eventually defines a flat (i.e. no hierarchical states) state-machine where only one mode can be active at a given point in time. Like also depicted in figure 4.20, those modes can be changed by a

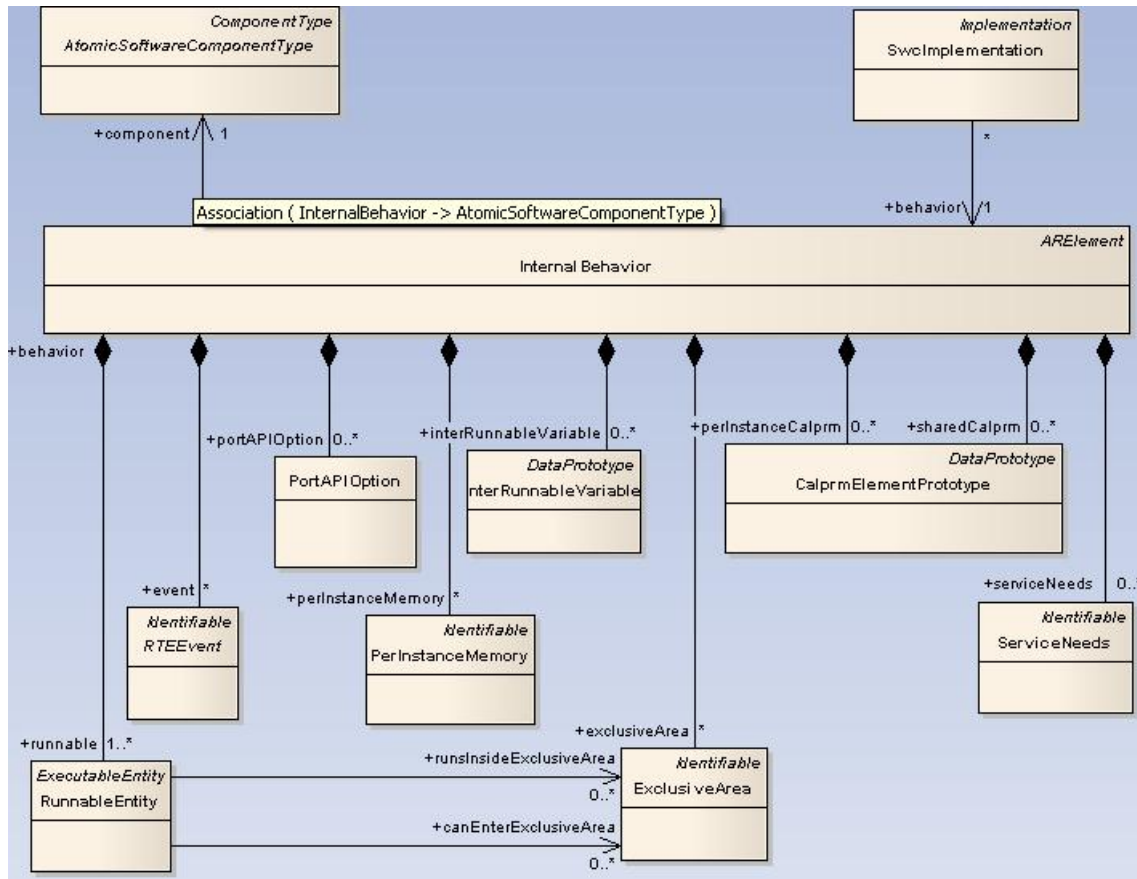


Figure 4.19.: AUTOSAR InternalBehavior of software components, from [12]

“ModeSwitchEvent”. Such modes can be used for prescribing sequences within a sender/receiver communication. By means of those sequences it is possible to define communication protocols between sender/receiver interfaces. Unfortunately, modes or states can not be specified in the context of client/server interfaces, which eliminates the possibility to define “state machines” or protocols for obligatory sequences of operation calls.

Furthermore AUTOSAR also defines some strategies in [15] for handling concurrency in terms of runnables which share memory:

- Mutual exclusion(mutex) with semaphores, which provide access to an exclusive resource that is used from within several tasks.
- Interrupt Disabling, during the run-time of runnable entities or at least for a period in time identical to the interval from the first to the last usage of a concurrently accessed variable in a runnable entity.

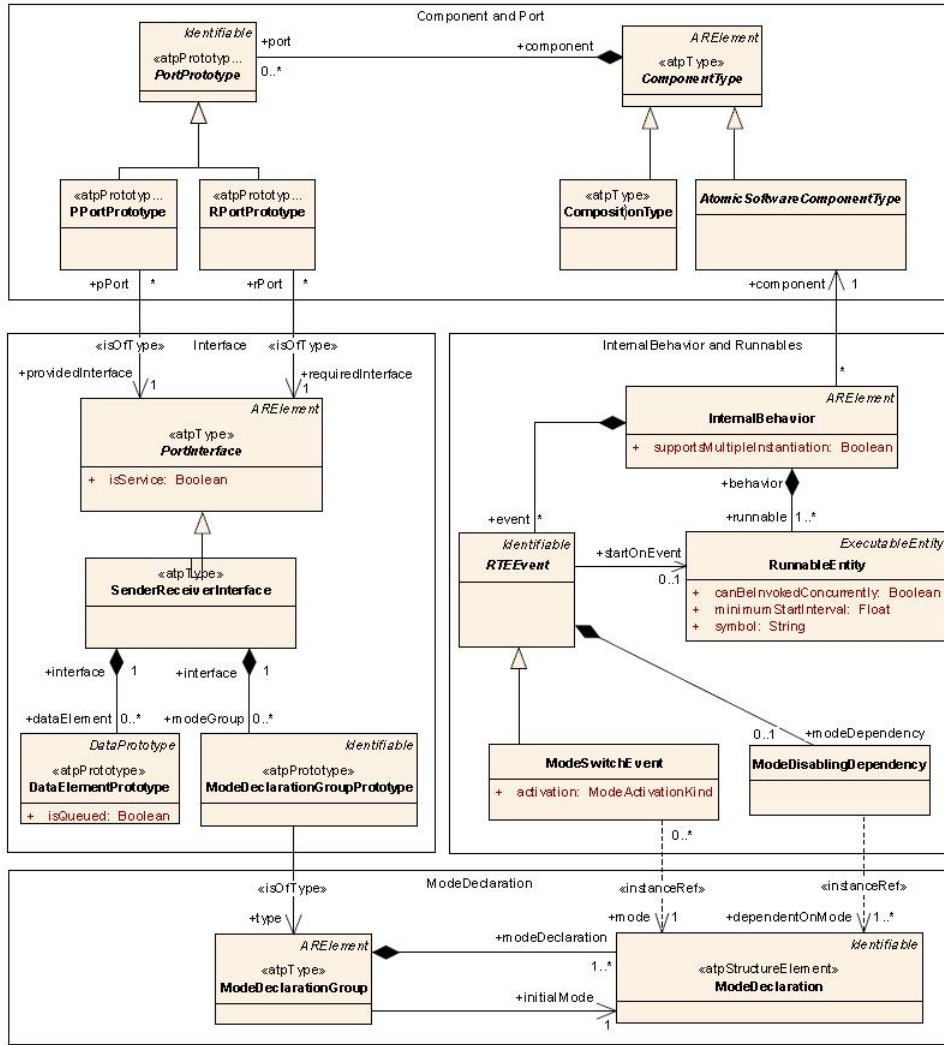


Figure 4.20.: Summary meta model excerpt related to modes, from [12]

- Priority Ceiling, which allows for a non-blocking protection of shared resources.
- implicit communication by means of variable copies, which means that for a concurrently used variable a copy is created on which a RunnableEntity entity can work without any danger of data inconsistency.

All these strategies can be implemented by the use of “ExclusiveAreas” and “InterRunnbalevariables” of the “InternalBehavior”.

In order to specify the resource consumption of an implementation, AUTOSAR provides a class called “ResourceConsumption”, which is depicted in figure 4.22. Each implementation must provide

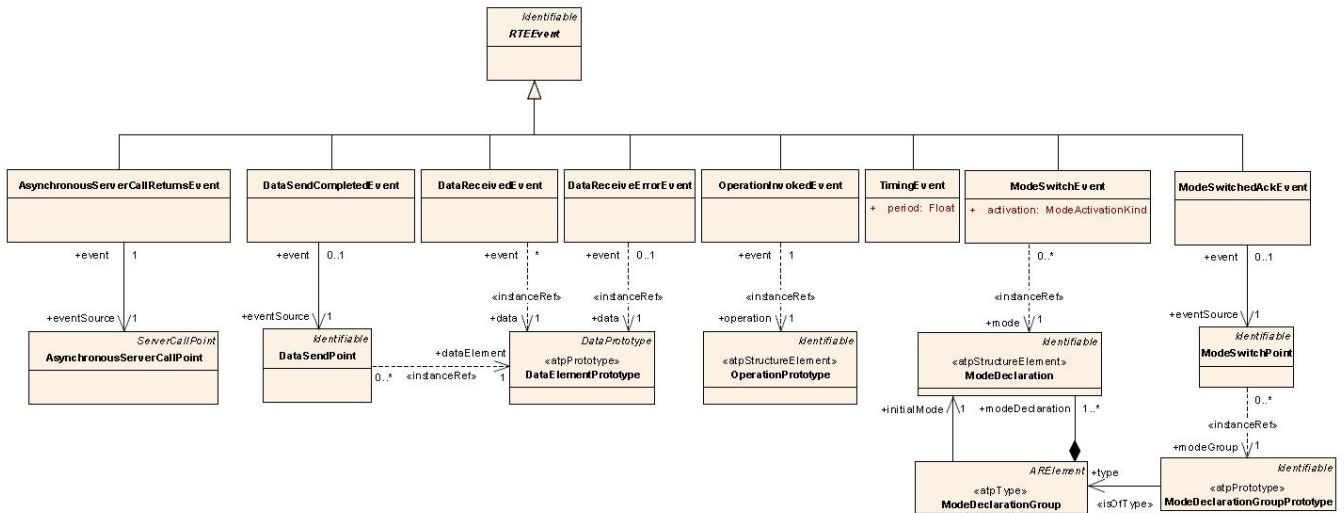


Figure 4.21.: Kinds of RTEEvents, from [12]

such a class to indicate its static and dynamic resource consumption. The resource consumption class can specify the used stack or heap usage as well as an execution time. The respective consumptions then can be associated with an executable entity, which can either be a RunnableEntity or a BSWModuleEntity. This enables the specification of memory or timing requirements on hardware or other software components.

Concerning the data type semantics figure 4.12 shows that AUTOSAR allows to specify ranges for numerical data types, like integer and float. Such ranges specify lower and upper limits to avoid overflows and underflows. Furthermore the figure shows that each primitive type may aggregate a so-called “SWDataDefProps” class. There the semantics are mainly described by units, which represent physical units and reference systems, and the aggregated “compuMethod”, which is used for the conversion of internal values into their physical representation and vice versa. However, “SWDataDefProps” [15] covers further various aspects as well:

- The binary structure of the data element
- The mapping/conversion to the data types in the programming language (or in Autosar)
- Invalid values for a data element
- If the data object is virtual - that means it is not directly in the ECU, then it can be described how the “virtual variable” can be computed from real ones.
- Code generation policies

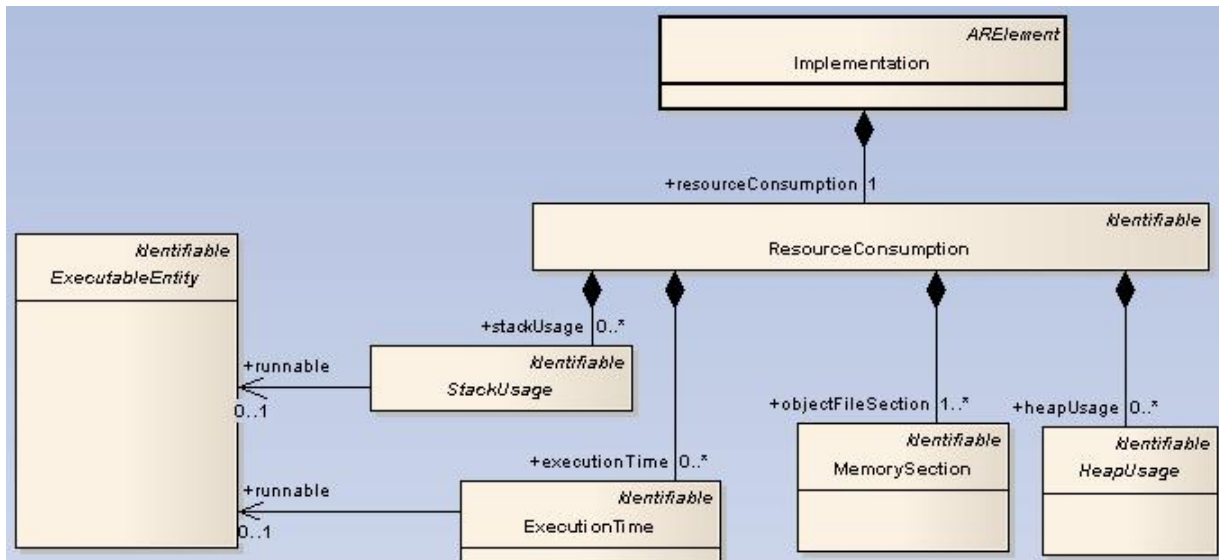


Figure 4.22.: AUTOSAR Resource Consumption: Overview, from [12]

Finally so-called “CompositeTypes”, which are build by the use of primitive data types, posses the described possibility to define the semantics of complex data types. Therefore it possible to compare two data types not only on a syntactical but also on a semantical level. However, because the semantical aspects only cover data types and not data themselves, it is not possible to describe semantics of data on a model level or among. For example: for a variable “temperature” typed with the data type float, it is not possible to specify whether the variable posseses the meaning of engine temperature, environment temperature, or else.

Benefits from AUTOSARs dynamic semantics:

The last section shows that on dynamic level the connection points can be described in a standardized and complete manner too. What kinds of inconsistencies can be avoided by these dynamic description elements, is described in the following:

- **Semantical inconsistencies** concerning the differing number representations or reference system inconsistencies can be avoided by the use of “SWDataDefProps” and “compuMethods”.
- **Application based inconsistencies** concerning data types ranges can be avoided by the use of the “Range” class, which is aggregated by all numerical data types. Furthermore, in the scope of “SenderReceiverInterfaces” it is possible to define “ModeDeclarations” and relationships between modes and events. This may avoid certain state violations as well as violations concerning the relationship between individual modes and input parameters.

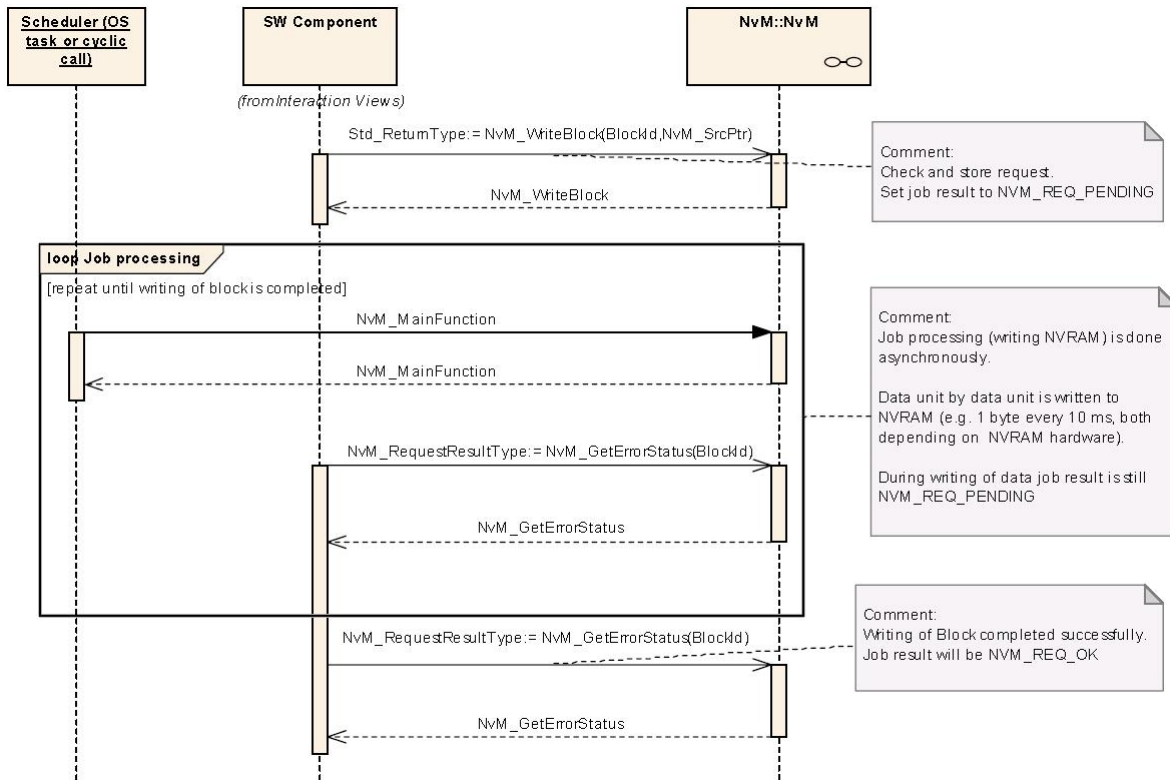


Figure 4.23.: NVM Sequence Diagram for WriteBlock operation, from [24]

- Pragmatical inconsistencies** concerning the concurrency behavior between components and their shared memory can be avoided by applying the common strategies of AUTOSAR which can be used to ensure the required data-consistency. “ModeDeclarationGroups” concern concurrency as well. By the means of “ModeDeclarations”, “RTEEvents”, and “RunnableEntities” it is possible to specify which of the runnables are allowed to start during a message sequence. This can also be seen in figure 4.20, where a “ModeDisablingDependency” is able to disable a “RTEEvent”, which otherwise would start a Runnable entity. By the means of runnable entities, which enable specifying read or write access, it is also possible to define access restriction on the extern resource “memory”. Furthermore, by means of “ServiceNeeds” and “ResourceConsumption” it is possible to avoid mismatching timing requirements on hardware elements.
- Missing pre-/postconditions or constraints for “operationPrototypes” and modes in terms of a “SenderReceiverInterface”. It is not possible to define constrained service parameter, obligatory relations between input parameters and internal states. This can lead to application based inconsistencies.

```

InternalBehavior NVRAMManager (
    // definition of associated operation-invoked RTE-events not shown
    // (it is done in the same way as for any SWC type)

    // section "runnable entities":
    RunnableEntity GetErrorStatus
        symbol "NvM_GetErrorStatus"
        canBeInvokedConcurrently = TRUE

    RunnableEntity SetDataIndex
        symbol "NvM_SetDataIndex"
        canBeInvokedConcurrently = TRUE

    RunnableEntity GetDataIndex
        symbol "NvM_GetDataIndex"
        canBeInvokedConcurrently = TRUE

    RunnableEntity SetRamBlockStatus
        symbol "NvM_SetRamBlockStatus"
        canBeInvokedConcurrently = TRUE

    RunnableEntity ReadBlock
        symbol "NvM_ReadBlock"
        canBeInvokedConcurrently = TRUE

    RunnableEntity WriteBlock
        symbol "NvM_WriteBlock"
        canBeInvokedConcurrently = TRUE

    RunnableEntity RestoreBlockDefaults
        symbol "NvM_RestoreBlockDefaults"
        canBeInvokedConcurrently = TRUE

    RunnableEntity EraseNvBlock
        symbol "NvM_EraseNvBlock"
        canBeInvokedConcurrently = TRUE

    RunnableEntity InvalidateNvBlock
        symbol "NvM_InvalidateNvBlock"
        canBeInvokedConcurrently = TRUE

    RunnableEntity SetBlockProtection
        symbol "NvM_SetBlockProtection"
        canBeInvokedConcurrently = TRUE

    // for each port providing the NvMService Interface:
    PortArgument {port= PS2, value.type=BlockIdType, value.value=2}
    ~
    PortArgument {port= PS<nm>, value.type=BlockIdType, value.value=<nm>}

    // for each port providing the NvMAdministration Interface:
    PortArgument {port= PAdmin<xx>, value.type=BlockIdType,
        value.value=<xx>}
    ~
    // end of section "runnable entities"
};

```

Figure 4.24.: NVM Behavior and Runnable Definition, from [24]

- Operation call protocols are missing in order to define an obligatory operation call sequence.
- No possibility to define semantics or meaning of data or functions on meta level M1.
- Individual pragmatistical inconsistencies concerning the absolute and relative timing behavior of components were not considered to await results of an actual project called TIMMO. This project will provide possibilities to enhance the timing specification of components

Table 4.3 summarizes these results. The table shows for each inconsistency (categorized in section 2.4.2), which can be avoided, the respective AUTOSAR meta class, which mainly is responsible for avoiding the inconsistency.

AUTOSAR Interfaces Example(dynamic):

To present a short example of how to apply AUTOSARs' dynamic interface semantics, the aforementioned NVM again will be used for demonstration of Basic Software Integration (see 2.3.2), compare to figure 4.14.

	avoidance possible	no special support
Syntactic Inconsistencies		
inconsistent data exchange format	SWDataDefProps	
inconsistent interfaces		X
inconsistent ports		X
inconsistent signatures		X
inconsistent data names		X
Semantical Inconsistencies		
numerical inconsistencies	SWDataDefProps	
language inconsistencies		X
reference system inconsistencies	SWDataDefProps	
Application-based Inconsistencies		
violation of states	partly by ModeDeclarationGroupPrototype	
violation of relations between states and parameter	partly by ModeDeclarationGroupPrototype	
restricted data ranges	Range	
Pragmatic Inconsistencies		
concurrency	ExclusiveArea, Inter-RunnableVariable, ModeDeclarationGroup	
access restriction on external resources	RunnableEntity	
timing requirements on hardware	ServiceNeeds, ResourceConsumption	
absolute timing requirements	TIMMO project	TIMMO project
relative timing requirements	TIMMO project	TIMMO project
communication pattern		X

Table 4.3.: Inconsistencies avoided by AUTOSARs' dynamic semantics

Beside a lot of requirements on the behavior of basic software functions which are specified within BSW specifications, some specifications also provide sequence diagrams to detail the behavior of BSW modules. Within the NVM specification [24] a lot of sequence diagrams prescribe the usage of functions specified by the aforementioned API. Figure 4.23, for examples, depicts a sequence diagram for a “WriteBlock” operation, whose syntax was specified in the API exemplified in figure 4.16. The sequence diagram shows the necessary sequence of function calls for an asynchronously performed “WriteBlock” request between a software component and the NVM controlled by the operating system. There are much more sequence diagrams (not only within the NVM specification), which prescribe the invocation semantics of API functions. These kind of specification is not a general applied AUTOSAR syntax element, but should additionally be used for describing the behavior of software components. Almost any basic software module uses the possibility of sequence diagrams, where the behavior can be standardized more than, for example, application software caused by the standardized character of basic software modules themselves. But also other software components should provide such descriptions in order to clarify their semantics.

Furthermore figure 4.24 depicts a simplified example for an “InternalBehavior” of the NVM. One can see that the “InternalBehavior” specifies “RunnableEntities”. Each runnable provides a symbol which indicates the invoked operation, which is specified by the API, and a tag which indicates whether the function can be invoked concurrently or not. Note that also the “WriteBlock” operation (the red bordered runnable in figure 4.24) from below is realized by a “RunnableEntity”. After all runables are specified, the “InternalBehavior” specifies some ports, which provide the service of the NVM.

4.1.4. Basic Conditions & EAST-ADL2.0

Beside AUTOSAR, the EAST-ADL2 standard also may avoid some inconsistencies by enhancing the basic conditions of development (compare to section 2.4.1). How far EAST-ADL2 is able enhance the actual situation, which was criticized within the survey, is again shown on the basis of section 3.2.1 in the following.

Completeness & Clearness

As aforementioned, EAST-ADL2 complements AUTOSAR with language elements to describe non-technical details and additional abstraction levels. To accomplish that, EAST-ADL2 provides a meta

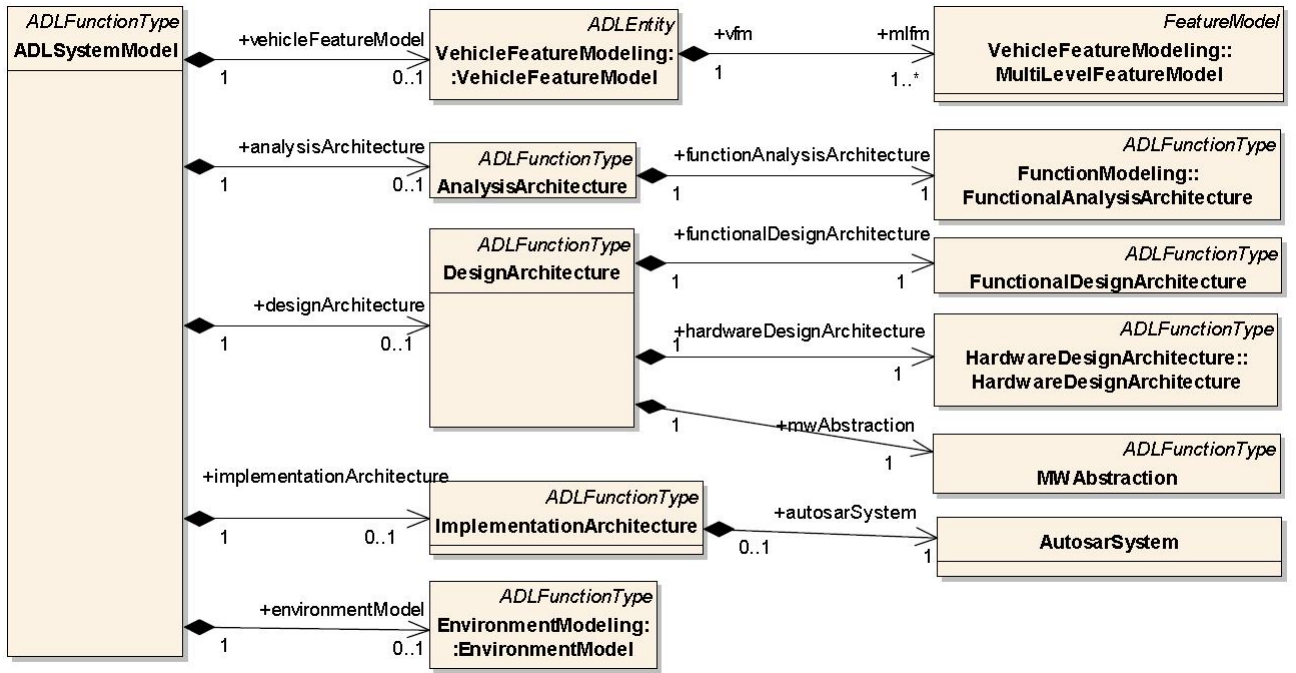


Figure 4.25.: EAST-ADL2 System Model: Overview, from [9]

model to specify necessary language elements and their semantics. In comparison with AUTOSAR, EAST-ADL2 only needs one main specification document to detail its meta model. The document, called “EAST ADL 2.0 Specification” [8], describes all artifacts, abstraction levels, and language elements, which are provided by the EAST-ADL2 meta model. However, here again the quantity of documents can not give any hints about the quality or completeness of EAST-ADL2.

Like above in section “Completeness & Clearness” of AUTOSAR (see 4.1.2), both levels the meta model level and the model level will be considered in terms of their completeness and clearness.

On meta model level

- **Completeness on meta model level:** As EAST-ADL2 shall complement AUTOSAR with additional abstraction levels, the EAST-ADL2 meta model must provide all syntax elements, which are necessary for a complete description of these abstraction levels. However, like in AUTOSAR, the EAST-ADL2 meta model is a large set of related syntax elements, which was defined by some human domain specialists. So it is hardly possible to verify that the meta model really provides all elements. For this reason, again just a rough overview about the meta model must suffice for the assumption, that it provides further abstraction levels and

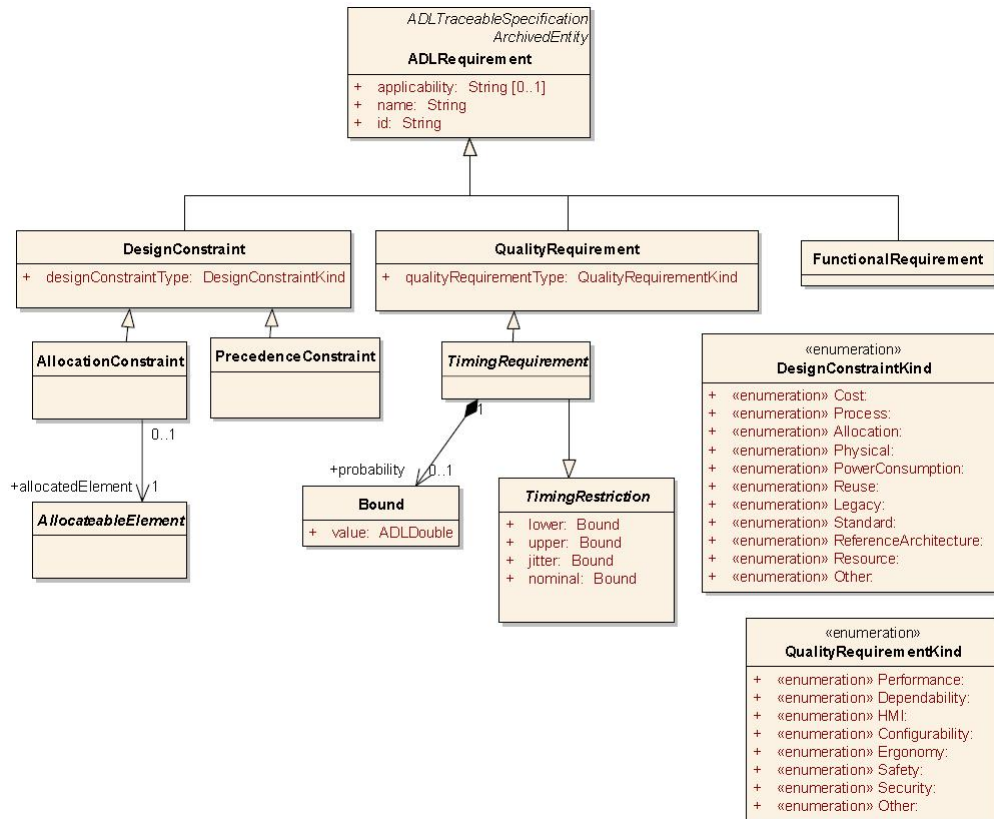


Figure 4.26.: EAST-ADL2 Requirements Modeling: Overview, from [9]

artifacts, whose syntax is sufficient to describe all relevant aspects of the respective artifact. Missing elements of individual artifacts will be shown by further evaluation projects in the future. Furthermore, the overview shall show how far AUTOSAR will be supplemented by the means of additional elements of EAST-ADL2.

The levels of abstraction as well as artifacts of EAST-ADL-2 were described in section 2.2.2. According to figure 2.5, figure 4.25 depicts a meta model overview of EAST-ADL-2, which shows the all-encompassing System Model of EAST-ADL. One can see, that a System Model aggregates the different abstraction levels and respective artifacts, which were described in section 2.2.2: The VehicleFeatureModel (VFM) which represents the most abstract level to describe perceivable features of a vehicle. The AnalysisArchitecture, which contains the FunctionalAnalysisArchitecture (FAA), to refine the VFM on Analysis level. The DesignArchitecture, which is subdivided into FunctionalDesignArchitecture, MWAbstraction, and HardwareDesignArchitecture, is used to refine the analyzed features of the FAA by some more implementation-oriented aspects. Furthermore, the System Model aggregates the Environ-

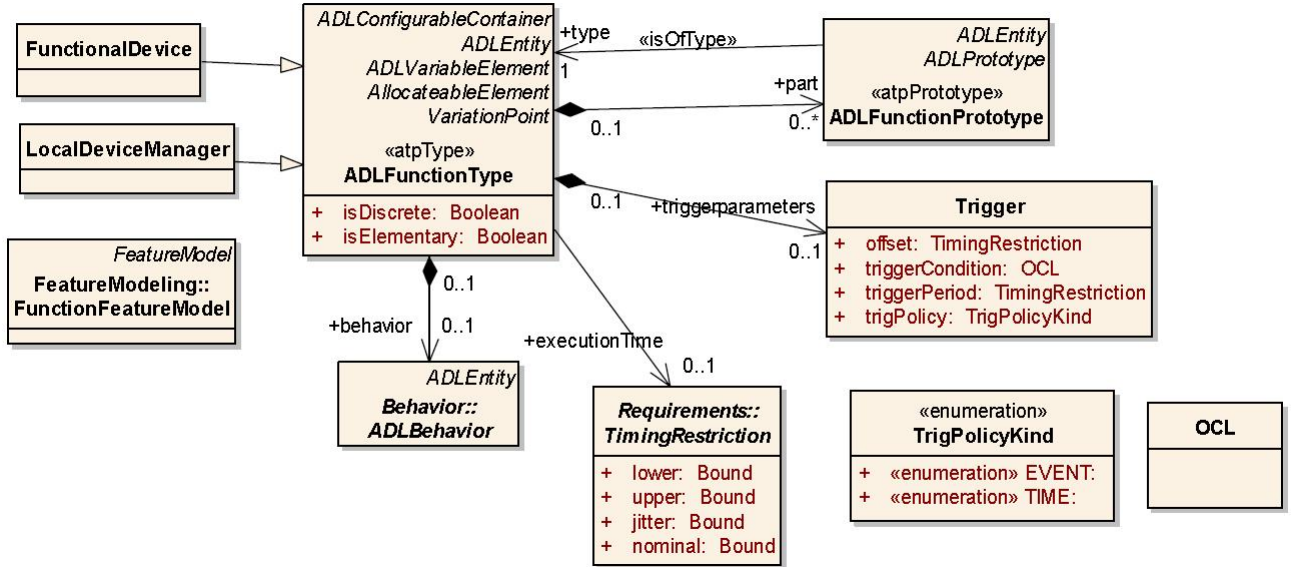


Figure 4.27.: Meta-model for the functional definitions of EAST-ADL2, from [9]

mental Model for features across all abstraction levels, and an ImplementationArchitecture, which is realized by AUTOSAR language elements for specifications on technical level. By the means of these artifacts, it is possible to represent an implementation architecture also on other abstraction levels, such as design, analysis, or requirements/features.

While on VFM level the core meta class “Feature” is used to describe all characteristics or traits that an individual product variant within a product line may or may not have, AUTOSAR syntax elements are used for specification on implementation level. By contrast, on FAA and FDA level another central concept, called “ADLFunctionType”, is used on both levels. “ADLFunctionTypes”, like depicted in figure 4.27 stand for individual “ADLFunctions”, which either refine features of the VFM or abstract from AUTOSAR entities. Beside the possibility to interconnect “ADLFunctions” by EAST-ADL connectors, which are depicted in figure 4.28, it is also possible to structure “ADLFunctions” hierarchically by the means of “ADLFunctionPrototypes”. Furthermore figure 4.27 shows that each “ADLFunction” may aggregate an “ADLBehavior” to specify the functions’ dynamic behavior.

But beside the possibility to model the structure and the behavior of functions, EAST-ADL2 provides language elements, which could be used, in principle, in any model on any abstraction level. These elements concern the specification of validation and verification (V&V), variability, and general requirements and can be found within the respective EAST-ADL meta model package.

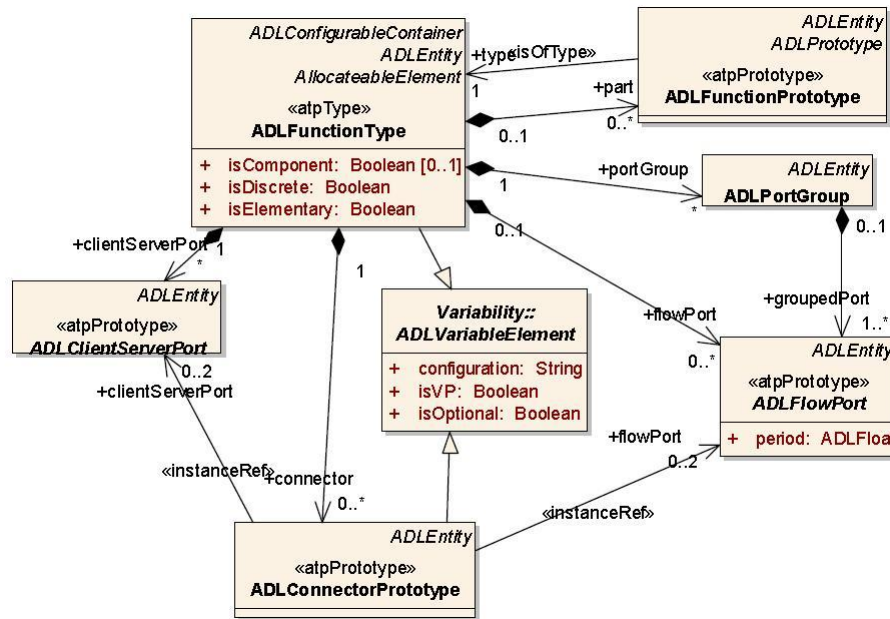


Figure 4.28.: Ports and connectors in the EAST-ADL2, from [9]

Concerning the V&V package it is impossible to introduce in the EAST-ADL2 a way to model all the objects that can be required by all available V&V techniques. For this reason EAST-ADL2 furnishes just the means for planning, organizing and describing V&V activities on a fairly abstract level and for defining the links between those V&V activities, the requirements that are checked by them and the objects modeling the system.

In the scope of variability EAST-ADL2 enables the possibility to express multiple variants of the same entity. Such entities, for example, may be features on VFM level, “FunctionTypes” on FAA or FDA level, different variants of implementations or artifacts and configurations in general.

The last EAST-ADL2 meta model package, which can be used across all abstraction level, concerns requirements. The requirements as part of existing EAST-ADL2 artifacts, are realized by the meta class “ADLRequirement”, see figure 4.26, and expresses a condition or capability that must be met or possessed by a system or system component to satisfy a contract, a standard, a specification or other formally imposed properties. A requirement may be a Design Constraint, a Quality Requirement, or a FunctionalRequirement. Design Constraints can be detailed by the use of individual “DesignConstraintTypes”, which indicate whether such a requirement concerns, for example, the costs, the process, reuse, or legacy of a model or model component. For quality requirements, different kinds of requirement can be specified

as well. By setting the “QualityRequirementType” of the meta class “QualityRequirement”, it is possible to specify requirements concerning, for example, performance, dependability, ergonomics, safety, or security of an EAST-ADL2 entity. Furthermore a quality requirement may also be a timing requirement, which especially can be used by “ADLFunctions” for defining their timing behavior in terms of end-to-end delays, periods and synchrony.

This rough set of syntax elements, which are more detailed by the EAST-ADL2 specification see [8], should show that EAST-ADL2 provides the possibility to specify automotive system on different abstraction levels. Like in AUTOSAR, no statement can be given about completeness of each artifact, as it would rise above the ability of one single person to understand any detail. Therefore it must be trusted in the expertises of several domain experts, which are developing the EAST-ADL2 standard, that EAST-ADL2 provides a complete or sufficient set of syntax elements.

- **Clearness on meta model level:** Even if it is possible to describe all abstraction levels and concerns completely by EAST-ADL2 syntax elements, it is important for developers to understand the meaning of syntax elements and the context within they are allowed to be used. For this reason the EAST-ADL2 standard specification [8] describes the EAST-ADL2 meta model in detail. This document details the meaning of syntax elements by the use of native text, in addition to the graphical representation of the meta model. Both the documents and the meta model are available to describe the semantics or meaning of EAST-ADLs’ abstract syntax and the relationships among its elements clearly. Consequently, clearness of the EAST-ADL2.0 meta model shall be ensured by a detailed specification document like in AUTOSAR.

The EAST-ADL2 meta model and its detailed description of syntax elements and their semantics, shall enable developers to specify abstract views on automotive systems. This again leads to the question if concrete specifications as model instances of the meta model are complete and clear.

On model level

- **Completeness on model level:** To model all obligatory artifacts completely in an ordered way, similar to AUTOSAR the EAST-ADL2 provides a methodology as well. For this reason section 2.2.2 has described the EAST-ADL2 methodology, which is aligned with a general V-Model. The character of the information in the upper parts of the V-Model are more or less

abstract and correspond to what is captured in the Vehicle, Analysis and Design levels of EAST-ADL2, like depicted in figure 2.7. The lower part of the V-Model represents the implementation level, which corresponds to an AUTOSAR modeled system within the EAST-ADL2 System Model. As EAST-ADL2 is aligned with that process and as the V-Model purports the order of these phases, it is recommendable to hold this order also during the usage of EAST-ADL2. By holding this order, it can be ensured that all artifacts are modeled step by step from the most general level down to the more detailed levels and that no artifact will be forgotten.

- **Clearness on model level:** Even if all artifacts are regarded during the development process, it is important to check whether all artifacts match the EAST-ADL2 meta model and whether constraints of the meta model are hold on the model level as well. This task can be undertaken by model checkers/analyzers, as in AUTOSAR. These model checkers and analyzers [31, 29], for example, can proof whether all relationships and attributes, which are prescribed by the meta model, are hold in order to ensure a certain level of completeness. Like in AUTOSAR model analyzers indeed are able to check allowed syntax elements, constraints and relationships, but it is not possible to check deeper semantics behind functions, data, or processes. On the one hand it is indeed possible to specify features, which are realized by ADLFunctions, to express what features individual functions are good for. This is an additional benefit concerning the semantical information in comparison with AUTOSAR definitively. However, individual descriptions such as ones for features or “ADLFunctions”, which are important to support the understanding of an entity, are realized by native text written from individual developers. Because machines are not able to interpret native text, the information are relevant for documentation more than for automating. However, also for documentation the textual descriptions have the drawback that they lead to misunderstandings and misinterpretations between multiple developers, like described in section 3.1.2.

All in all EAST-ADL2 provides abstraction levels and artifacts for analysis and design of automotive architectures beside their technical details. By the means of different views and additional descriptions, which focus other aspects of the entire system while they abstract from individual details, the understanding for an entire system can be enhanced clearly and contexts within the system can be clarified better than on implementation level. In addition, EAST-ADL2 adds variant building, validation and verification, and functional an/or non-functional requirements to AUTOSAR, so that EAST-ADL2 not only provides abstraction levels for the technical level of AUTOSAR, but also additional information about a system or system components. Unfortunately, EAST-ADL2 has the same drawback

like AUTOSAR. This drawback concerns the textual descriptions which are used to detail certain semantical information of EAST-ADL2 entities. Because native text as basis for a later implementation may lead to different interpretations and misunderstandings mostly, inconsistencies between software components on implementation level are preassigned.

Consistency

Consistency between artifacts, tools, and divisions is important not only in the scope of AUTOSAR but also of EAST-ADL2. For this reason the following section will analyze EAST-ADL2 to find out, how it supports the consistency for artifacts, tools, and divisions.

Consistency between artifacts:

- Like in AUTOSAR the consistency between meta model level and model level has to be ensured by applying the EAST-ADL2 profile within a modeling tool. This again restricts general modeling elements to an EAST-ADL2 conform syntax, which can be verified by model analyzers/checkers on the model level. Furthermore, the profile mechanism ensures that all specifications/models are modeled by the same syntax.

EAST-ADL2 indeed provides a relationship mechanism to ensure continuity between artifacts, as described further below in section 4.1.4. This can hold consistency between artifacts too. However, this mechanism only supports developers to indicate dependencies between model elements for their understanding and view/model integration (compare to section 2.3.3). The mechanism only shows dependencies, it is not able to proof whether model elements are kept consistent. Therefore applying the profile is the single explicit mechanism of EAST-ADL2 to hold consistency between artifacts within a tool generally. Unlike AUTOSAR, which prescribes certain requirements for tool functionality and modeling rules, the EAST-ADL2 meta model is just intended to provide syntax elements, which can be used within any modeling tool. For this reason EAST-ADL2 left consistency checks for artifacts to the individual features of modeling tools. Therefore, in order to hold artifacts more consistently, EAST-ADL2 also should introduce some modeling guide lines which ensure consistent naming, structuring, and modeling between multiple developers.

Consistency between tools:

- Due to the wide-spread development, EAST-ADL2 artifacts or models also have to be exchanged between different tools, developers, and divisions just like AUTOSAR models. Unfortunately, EAST-ADL2 does not provide any specifications, requirements or guide lines for the interaction between tools like AUTOSAR. In particular, EAST-ADL2 does not describe an exchange format for models, as it is done by the Model Persistence Rules [14] of AUTOSAR, see section 4.1.2. Because almost any modeling tool however provides its own import/export mechanism, it is still possible to import or export EAST-ADL2 models on basis of the same EAST-ADL2 profile. Because EAST-ADL does not make any instructions for the format, the exchange format depends on the used tool, which may differ from party to party. This represents a critical drawback for EAST-ADL2, because due to the non-specified exchange format any exchange format can be used. So it is probable that most of these formats do not match, which hampers the exchange of artifacts and leads to expensive reworking, loss of information, and inconsistencies.

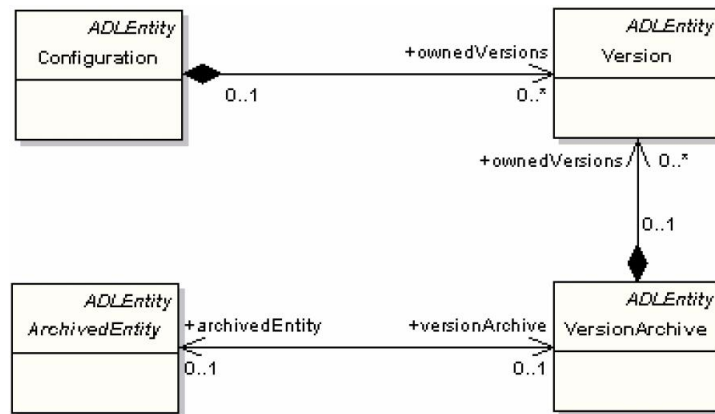


Figure 4.29.: EAST-ADL2.0 Support, from [8]

Consistency between divisions:

- EAST-ADL2 provides, similar to AUTOSAR, a small package to support consistent matching of models between divisions. This package, called “Support” is depicted in figure 4.29 and shows commonality with the “AdminData” package of AUTOSAR in figure 4.8. The “Support” package also enables to specify version information for individual so-called “ArchivedEntities”, which mark instances of other EAST-ADL2 entities as elements for which a version management will be established. Therefore it is possible to provide artifacts with version information, before the “Configuration” meta class is able to gather version-controlled artifacts for one build or configuration.

Unlike AUTOSARs' "AdminData", the "Support" package of EAST-ADL2 does not provide annotations for modifications of individual artifacts. Thus it is indeed possible to define version-controlled configurations in order to ensure matching artifact versions by the means of EAST-ADL2. But concerning modifications and a cross-division development, different parties rely on the help of individual change or version management system, which may differ from party to party.

All in all one can say that EAST-ADL2 does not support consistency efficiently. It would be advisable to define some obligatory consistency requirements or to standardize modeling rules and/or persistence rules like in AUTOSAR. By the means of such standardization it could be ensured that all parties handle consistency in the same manner in order to avoid individual solutions, which must be adapted when information is exchanged.

Continuity & Traceability

Because one EAST-ADL2 system model could hold all artifacts of EAST-ADL2 or AUTOSAR, it can be ensured that all necessary information is available for developers' work. Just because EAST-ADL2 introduces several abstraction levels to represent automotive architectures and in order to support developers to orient oneself within this large set of artifacts and abstractions, EAST-ADL2 provides a mechanism to ensure continuity and traceability.

EAST-ADL2 supports continuity and traceability by a large set of relationships to interconnect ADL entities, whereas each relationship possesses its own semantics. Figure 4.30 shows the meta model excerpt for EAST-ADL2 relationships, which depicts five relationships:

- The "ADLDeriveReq" relationship, which signifies a dependency relationship in-between two sets of ADL requirements.
- The "ADLRealization" relationship, which signifies realization relationship in-between two sets of model elements, one representing a specification (the supplier) and the other represents an implementation of the latter (the client).
- The "ADLRefine" relationship, which signifies a dependency relationship in-between an ADL requirement and an ADL entity, showing the relationship when a client ADL Entity refines the supplier ADL requirement.

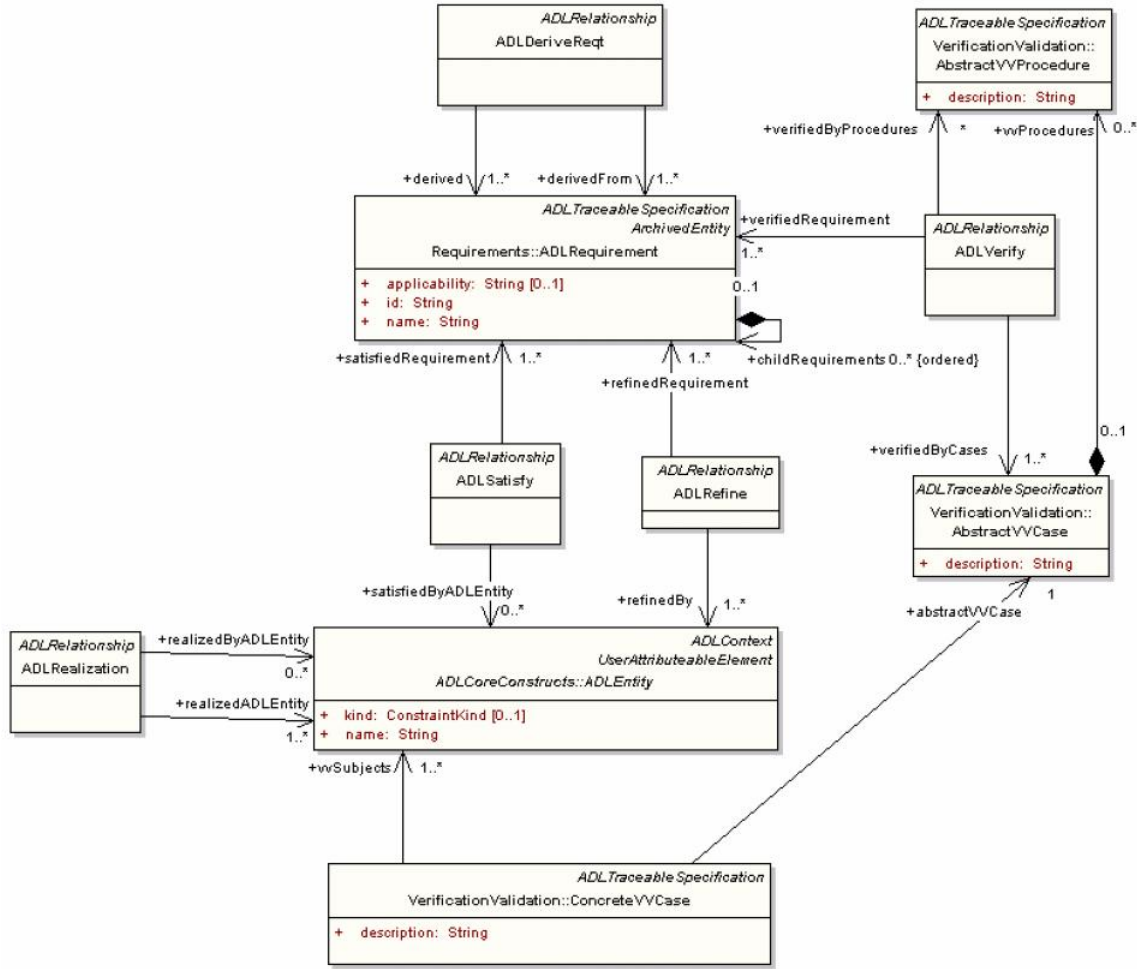


Figure 4.30.: EAST-ADL2.0 Relationship Modeling, from [8]

- The “ADLSatisfy” relationship, which signifies relationship in-between ADLRequirement and satisfying ADL entity, showing the relationship when a satisfied supplier ADL Requirement is satisfied by the client ADL entity or satisfied by the client AUTOSAR element.
- The “ADLVerify” relationship, which signifies a dependency relationship in-between an ADL requirement and a V&V Case, showing the relationship when a client V&V Case verifies the supplier ADL requirement.

All these relationships can be used either to interconnect ADL entities between abstraction levels or to add or aggregate additional information. Therefore they represent a well-established mechanism for view and model integration (compare to section 2.3.3) in order to ensure continuity between various artifacts and abstractions.

For example, figure 2.10 shows the interplay between the EAST-ADL2 design level and AUTOSARs' implementation level, realized by ADLRealization relationships. One can see that ADLFunctions on design level may be interconnected with AUTOSAR Runnables of an application software component(see section 4.1.3) on implementation level directly. Like here, also ADLFunctions may be interconnected between design and analysis level or with features on vehicle level. However, not only for ADLFunctions, Features or Runnables it is possible to interconnect them, but also for requirements or variants it is possible to ensure continuity between different abstractions. Furthermore by the means of relationships it is possible to trace back individual implementations to other abstraction levels and development processes phases. Also an incremental development is supported by EAST-ADL2 relationships, because solutions need not to be specified from scratch. Instead of that, information can be gathered on one abstraction level to represent rough overview about the architecture, before these information can be refined on the next abstraction level below.

In conclusion EAST-ADL2 provides the availability of all specifications concerning the system not only on technical level. As not only the dependencies between artifact elements are prescribed by the EAST-ADL2 meta model, but also the dependencies among abstraction levels are defined by relationships clearly, AUTOSAR can be completed by non technical details using the EAST-ADL2 in a continuous manner. By the means of additional EAST-ADL2 artifacts which explain the technical design solutions, it is possible to understand individual design solutions of AUTOSAR better than without EAST-ADL2. So, the drawback which was mentioned in section 4.1.2 concerning the missing artifacts of AUTOSAR can be resolved by using the EAST-ADL2.

Integration of all disciplines

As mentioned in section 4.1.2, AUTOSAR misses disciplines from the levels before implementation or beside the technical focus. However, even such levels, like requirements, analysis or design, are provided by the EAST-ADL2 artifacts. A requirements engineer is supported by the feature modeling package on vehicle level as well as by the requirements package. By the means of analysis and design level it is possible to support engineers disciplines, which concern either software, hardware or mechanics, on higher levels of abstraction too. Furthermore, engineers are supported to model a global environment model, which concerns any level abstraction. By the means of such an environment model it is not only possible to specify the system itself, but also the external basic conditions in which the system must be integrated.

But EAST-ADL2 also supports further disciplines: Firstly it provides means to support the discipline of Validation and Verification by furnishing the means for planning, organizing and describing V&V activities on a fairly abstract level, like mentioned in section 4.1.4. Secondly the variability package supports product line engineers to describe different variants and differences among products and product families. The advantage, which comes along with EAST-ADL2, is that V&V descriptions as well as variability descriptions can be interconnected with the system or system components directly via relationships within the model, as described within the last section.

However, although EAST-ADL2 supports a lot of disciplines, the EAST-ADL, like AUTOSAR, does not support the discipline of an integrator explicitly but implicitly. Additional abstractions or views enable integrators to get a better appreciation for the overall system, because integrators can leave away individual details to analyze the overall structure more than technical details of a single component. For this reason, virtual integration on implementation level, which is supported by AUTOSARs technical view (see section 4.1.2), will be complemented by additional integratable views (compare to view integration within section 2.3.3) as well as further possibilities for virtual integration on higher levels of abstraction. Furthermore the continuity between all artifacts up until vehicle level, enables integrators to understand why individual solutions were chosen. Because features/requirements or ADLFunctions must be satisfied by system components on implementation level, an integrator is able to understand the development process better which leads to the respective implemented technical solution. Because abstractions of the entire system are available much earlier on analysis and design level than on implementation level, the early system overview enables integrators to prepare for integration much earlier as well.

Maintenance of artifacts

EAST-ADL2 primarily supports the maintenance of artifacts like described before in section 4.1.4. By the means of the support package, which is depicted in figure 4.29, it is possible to support version management systems by version and configuration information, which can be annotated with ADL artifacts. Moreover, the variability package of EAST-ADL2 supports maintenance of artifacts by grouping system components in accordance with individual products and product lines. Thereby it is possible to maintain these variants within one model and without using additional management systems for maintaining product line specifications.

Because EAST-ADL models can be split into multiple files like AUTOSAR models, these files have

to be managed as well. Unfortunately EAST-ADL2 also provides no guide lines or means to support the these kind of maintenance. There are no file structures prescribed and also changes remain on the side of additional change and version management systems, which leads to individual solutions and inconsistencies. For this reason, EAST-ADL should follow section 4.1.2 by defining recommendations for change management and some file structure rules.

Assignment of roles and responsibilities

EAST-ADL2 also does not assign roles and responsibilities and leaves it to the respective division how to apply EAST-ADL2. It is indeed possible to derive individual roles from the aligned V-model and its respective artifacts, but there are no explicit guide lines given. For this reason, divisions can apply EAST-ADL2 individually, which means an advantage because any division is able to integrate the ADL into its own development process. On the other side it will lead to inconsistent processes and role allocations, which lead to the same drawback like within AUTOSAR (compare to section 4.1.2). Therefore EAST-ADL2 should at least provide a framework of roles and responsibilities, which can or should be applied by any division in order to avoid misunderstandings and forgotten roles or responsibilities.

Conclusion EAST-ADL2 Support to enhance Basic Conditions



Figure 4.31.: EAST-ADL2 Support to enhance the basic conditions

In conclusion figure 4.31 summarizes all results concerning basic conditions of development, which can be enhanced by EAST-ADL2. Thereby the arrows indicate how far AUTOSAR provides support

for a current critical point. Unfortunately, there is also no measurement, which measures enhancements exactly. For this reason the grade of enhancement was assessed subjectively on the basis on variety and the helpful character which is provided by EAST-ADL2s' specifications.

In comparison with AUTOSAR, see figure 4.10, also EAST-ADL2 supports completeness and clearness for artifacts very well, while roles and responsibilities are not supported at all. Other points are also supported by EAST-ADL2 in the same intensity like AUTOSAR. Thereby all points are not supported redundantly, but according to the respective concerns of each standard. Only one critical point concerning the consistency of artifacts was left open from EAST-ADL2. The red arrow at this point indicates, that there is need for strong action in order to enhance consistency support not only for AUTOSAR artifacts, but also for EAST-ADL2 artifacts. Furthermore, such as AUTOSAR, EAST-ADL2 also should extend the yellow or orange colored points in order to simplify integration further more.

So, also EAST-ADL2 should establish guide lines or best practices concerning amongst others the aforementioned six points. Such rules should describe the best usage of EAST-ADL2 concerning the aforementioned points and its language elements or artifacts in general. Although simple rules are not a panacea, it would be a first step into enhancement.

4.1.5. Concrete Inconsistencies & EAST-ADL2.0

Beside reasons for inconsistencies, there are also some concrete inconsistencies (compare to section 2.4.2), which may be avoided by using existing EAST-ADL2 concepts too. If the EAST-ADL2 standard would be applied generally, the common base could avoid most of the problems, faults, or failures. On the basis of the points, which were identified in section 3.2.2, the following will show how far EAST-ADL2 is able to avoid some concrete inconsistencies, which were criticized within the survey.

Static semantics of interfaces

EAST-ADL2 language elements to describe static semantics of Ports & Interfaces:

The EAST-ADL2 only provides a small set of syntax elements to describe the static semantics of interfaces. Each ADLFunction may provide "ADLClientServerPorts" and/or "ADLFlowPorts", which are interconnected via "ADLConnectorPrototypes", as depicted in figure 4.28.

ADLClientServerPort metaclass is an abstract port for client-server interaction, where an ADLClientPort represents the client side and an ADLServerPort represent the server side in a client-server interaction.

On the other side “ADLFlowPorts”, which can be used for sender-receiver interaction, are interaction points through which input and/or output of items such as data, material or energy may flow. There are three meta classes derived from it: The ADLInFlowPort metaclass, which denotes a port that requires one data. The ADLOutFlowPort metaclass, which is a port that provides one data. And the ADLInOutFlowPort metaclass is a port with both provided and required interfaces.

ADL ports allow to define the connection points between components/function by providing information about the applied communication pattern (Client/Server or Sender/Receiver) at the same time. However, further syntactic or static details are not supported by the EAST-ADL2 directly, but by the additional means of AUTOSAR, these information can be refined on implementation level of EAST-ADL2.

Dynamic semantics of interfaces

EAST-ADL2 language elements to describe dynamic semantics of Ports & Interfaces:

Concerning the dynamic semantics of interfaces, EAST-ADL2 primarily provides a behavior package in order to describe the dynamics of the system or system components. On the one hand this package enables the definition of use cases by the means of “ADLUseCases”. ADLUseCase, which purpose is to define expected characteristics in a user-oriented manner, specifies a set of actions that the associated entity performs. On the other side the package contains a so-called “ADLBehavior” meta class, which can be aggregated by an ADLFunction like depicted in figure 4.32. Because “ADLBehavior” is an abstract meta class, there are two further concrete behavior meta classes for detailing the ADLFunctions’ behavior:

The “NativeBehavior” is a specialization of ADLBehavior and the UML2 StateMachine. It is a placeholder for a behavioral definition that is recognized by EAST-ADL2 compliant tools.

The ExternalBehavior is a specialization of ADLBehavior too. It represents behavior defined in an external tool, such as Simulink or Statemate. However, it is merely a placeholder with the purpose of containing information about and links to the external behavioral model.

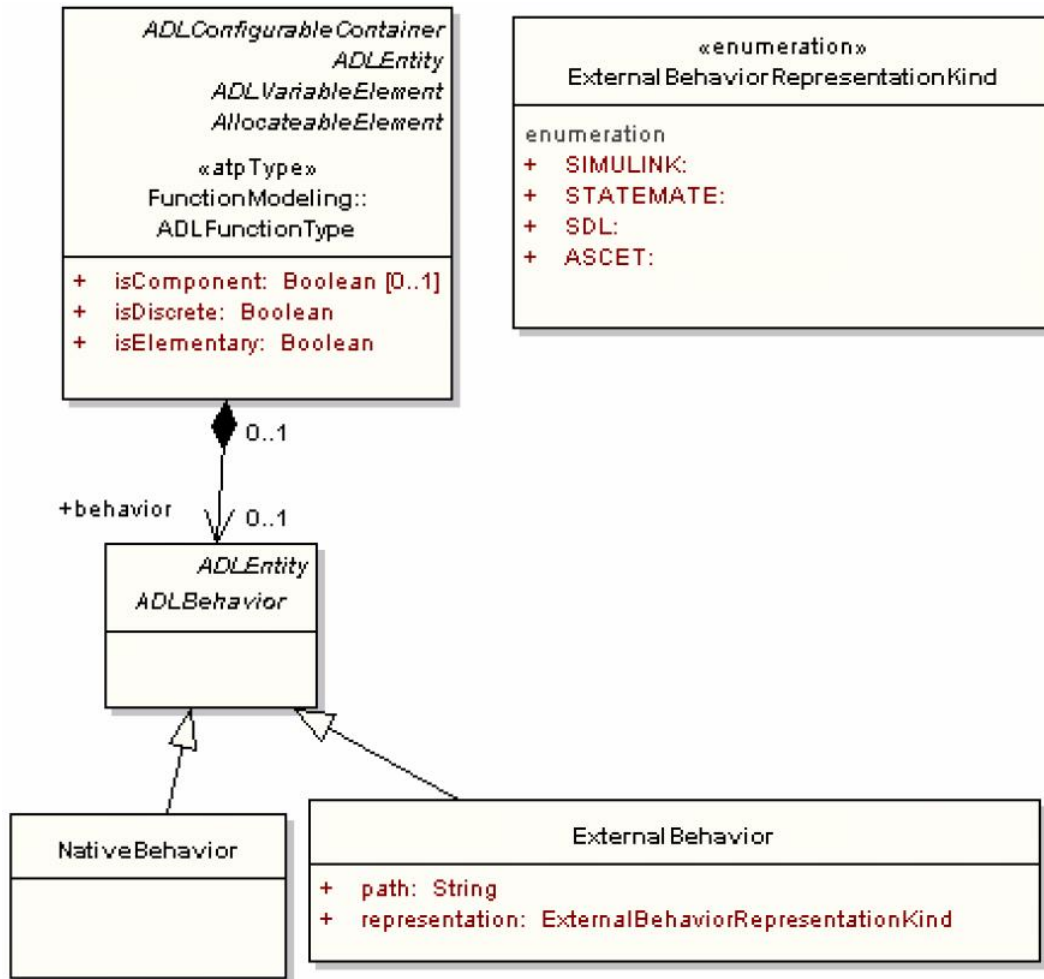


Figure 4.32.: EAST-ADL2.0 Behavior Constructs, from [8]

Additionally, in order to refine the dynamic specification of functions, EAST-ADL provides the definition of timing requirements and triggers.

Timing requirements, like depicted in figure 4.33, are for defining requirement on end-to-end delays, periods and synchrony. The figure shows that timing requirements may concern individual quality requirements, which is indicated by the means of the quality requirement type. Furthermore each timing requirement may specify some timing restrictions, which specify bounds on system timing attributes, i.e. end-to-end delays, periods, etc. Such timing requirements can be refined in order to define synchronization between ports or to define some delay segments which constrain the delay between two instants of data creation. Delay segments are used for defining requirements on end-to-end delays and timing chain segments.

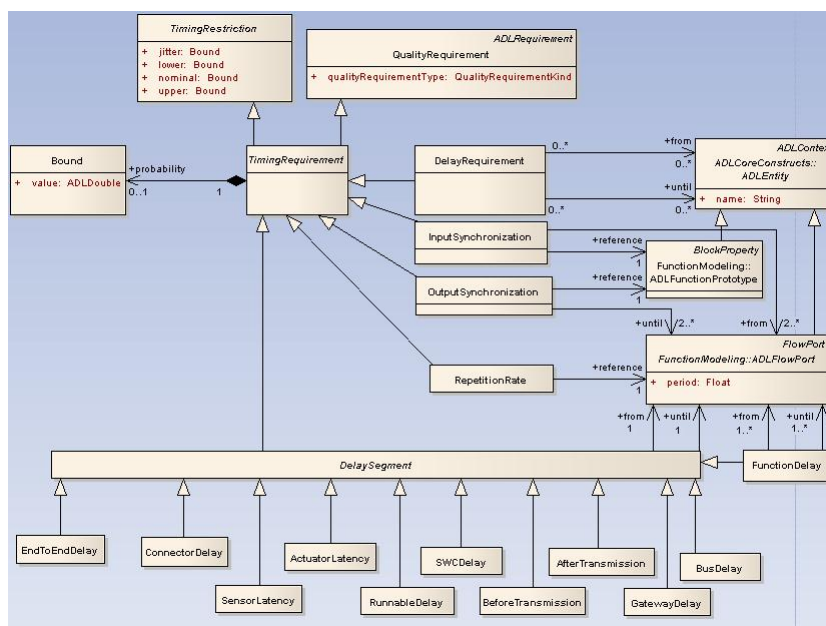


Figure 4.33.: EAST-ADL2.0 Timing Requirements, from [9]

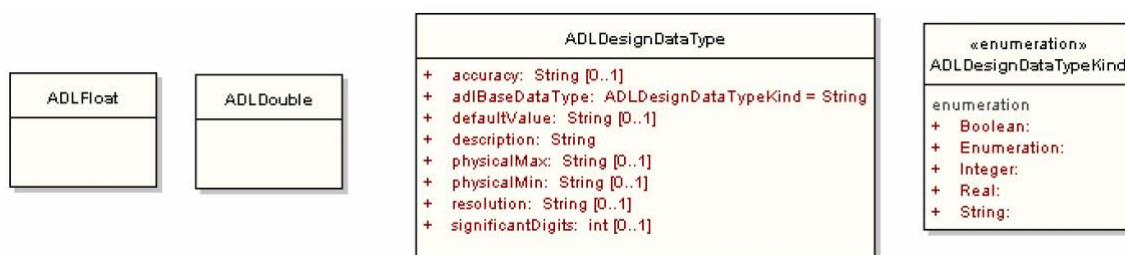


Figure 4.34.: EAST-ADL2.0 Data Types, from [9]

The trigger class contains the trigger parameters necessary to define the execution of the containing ADLFunctionType, as depicted in figure 4.27. The containing ADLFunction may be triggered according to the attributes of the Trigger class. By the means of the trigPolicy attribute of the trigger class is possible to define whether communication is event or time based. For example, such trigger policies enable to specify whether functions are called periodic (trigPolicy=TIME), or whether functions are event-triggered (trigPolicy=EVENT). On the other side, trigger specify OCL expressions that allow release of the ADLFunctionType only if it evaluates to TRUE. However, OCL is a standardized language, which is not aligned with the automotive domain, in order to define constraints. Therefore it represents a good enhancement in comparison with AUTOSAR, but it can not be assumed that all necessary restrictions can be expressed via OCL.

EAST-ADL2 also provides a good basis for data type semantics concerning ADLFlowPorts and sender-receiver interaction. By the means of the meta classes ADLDesignDataTypes, ADLFloat and ADLDouble, which are depicted in figure 4.34, it is possible to use predefined data types. In order to specify the semantics of data types, the ADLDesignDataType aggregates some attributes for detailing the data type. Such data type information can be used and refined by AUTOSARs syntax elements on implementation level.

Benefits from EAST-ADL2s' static and dynamic semantics:

The last section shows that on dynamic level EAST-ADL2 provides some syntax elements to substantiate further information, which complement knowledge on implementation level to avoid more inconsistencies than by using AUTOSAR exclusively. What kinds of inconsistencies are able to be avoided by these dynamic description elements, is described in the following:

- Semantical inconsistencies between specification can be avoided by the means of the standardized meta model of EAST-ADL2, as in AUTOSAR. As all available syntax elements are defined clearly, the meaning of elements will not be misinterpreted.
- The avoidance of application based inconsistencies concerning states and state relations may be supported by the usage of ADLFunctions' triggers and their respective triggerConditions. By the means of these triggerConditions it is possible to define conditions for component states via OCL. Only if an OCL constraint can be evaluated to TRUE, an ADLFunction will be executed. As OCL also provides pre- and postconditions, it is also possible to define such conditions for function on analysis and design level. Beside that, the EAST-ADL2 data type specification also allows to define physical data type ranges in order to restrict allowed ranges for interaction of two or more components.
- Pragmatical inconsistencies concerning the absolute and relative timing behavior of software or hardware components can be specified by the means of timing requirements. The communication pattern can also be predefined by the semantics of EAST-ADL2 ports. Moreover, concurrency behavior of components and access restriction can be defined by some OCL constraints rudimentarily.
- EAST-ADL2 also just provides textual descriptions to clarify individual meanings of components and data. This means a critical drawback, as native text again may lead to misunderstandings and misinterpretations, like in AUTOSAR.

Table 4.4 summarizes these results. The table shows for each inconsistency (categorized in section 2.4.2), which can be avoided, the respective EAST-ADL2 meta class, which mainly is responsible for

	avoidance possible	no special support
Syntactic Inconsistencies		
inconsistent data exchange format		X
inconsistent interfaces		X
inconsistent ports		X
inconsistent signatures		X
inconsistent data names		X
Semantical Inconsistencies		
numerical inconsistencies		X
language inconsistencies	meta model	
reference system inconsistencies		X
Application-based Inconsistencies		
violation of states	partly by OCL triggerConditions	
violation of relations between states and parameter	partly by OCL triggerConditions	
restricted data ranges	ADLDesignDataType	
Pragmatic Inconsistencies		
concurrency	partly by OCL triggerConditions	
access restriction on external resources	partly by OCL triggerConditions	
timing requirements on hardware	ADLRequirement	
absolute timing requirements	TimingRequirements	
relative timing requirements	TimingRequirements	
communication pattern	ADLFlowPort, ADL-ClientServerPort	

Table 4.4.: Inconsistencies avoided by EAST-ADL2s' dynamic semantics

avoiding the inconsistency.

4.1.6. Conclusion

The last five sections have shown that EAST-ADL2 and AUTOSAR are able to enhance the actual situation of automotive software engineering in many respects.

By the means of the standardized character of both specifications the basic condition of development can be enhanced and some concrete inconsistencies can be avoided. Like mentioned above, the meta models, their detailed documentations, and their respective methodology are able to ensure that specifications are defined completely and clearly as well. While meta model and methodology ensure that no artifact or specification element will be forgotten, the detailed documentations will avoid misunderstandings or misinterpretations concerning syntax elements, which are used for specification of concrete automotive systems.

Moreover, both standards enable structuring of the entire system and its components well. Packaging mechanisms and building hierarchies enable developers to structure components and artifacts, whereas all components and artifacts are interconnected via relationships. Thereby it is possible to trace correlations within one single model, which can be created as result of multiple integratable models. EAST-ADL2 even provides relationships by which both standards can be integrated into a whole, so that all advantages, which come along with one respective standard, can be brought together. Because all artifacts or components are able to be integrated into a whole, all information can be made available for developers. However, information are not only available, relationships between all artifacts allow to contextualize information. By the means of relationships information of interconnected components as well as information of other abstraction levels are available and interconnected, that it is possible to switch between all information continuously. All these benefits, which come along with the complete availability of interconnected information, will avoid missing of information as well as misunderstandings, which can be prevented because correlations are presented superiorly.

Furthermore, while EAST-ADL2 only prescribes the meta model or UML2 profile to handle consistency between specified artifacts, AUTOSAR even enhances consistency in addition to the meta model. By the means of predefined requirements and additional specification documents, a consistency mechanism, which should be hold by any party, is defined in order to ensure that all parties do handle consistency between artifacts and tools in the same manner. In addition to that, both

standards indeed provide a rudimentary consistency support for different divisions and parties, but further standardization will be necessary to prevent individual solutions concerning, for example, mismatching file structures, management of artifact changes, or the cross-divisional alignment of baselines or releases.

Different disciplines are supported by EAST-ADL2 and AUTOSAR as well. Both standards together enable any kind of software or hardware engineer to specify components on different levels of abstraction. However, also requirements, testing, or variability disciplines are integrated and are able to interact with any other discipline via “one” standard. Especially integration is well-supported by these two standards implicitly. As models provide an overview about all abstraction levels, details, and/or disciplines, an integrator is supported by a maximum possible set of information, which is necessary to understand the entire system for integration.

But though EAST-ADL2 and AUTOSAR support as many disciplines, they do not provide any framework or guide lines for prescribing roles or responsibilities of disciplines or developers within a development process. On the one hand this enables each division to integrate both standards into existing structures without great adaptations. On the other hand, due to that lack, each division will distribute roles and responsibilities in an individual manner, so that inconsistencies between divisions and neglected responsibilities are preassigned. For this reason it would be advisable at least to provide a framework for this concern. Because such a framework is currently under development, as described in section 2.3.4, the results and experiences from that project should be awaited to make detailed statements or recommendations.

All these benefits, which come along with EAST-ADL2 and AUTOSAR, enhance the basic conditions which were criticized from interviewees in the survey, like described in chapter 3. EAST-ADL2 in combination with AUTOSAR provide sufficient means to enhance the actual development situation. This situation is responsible for a lot of reasons (compare to section 2.4.1), which cause inconsistencies between specifications and/or codes hampering a later integration. By enhancing the basic conditions, multiple reasons for inconsistencies can be avoided before a project starts. Although further standardization of additional guide lines or best practices, which would blow the scope of this thesis, could enhance the basic conditions further more. The two standards represent a better basis for development in comparison with today's applied techniques.

However, not only the basic conditions themselves are enhanced by AUTOSAR and EAST-ADL2. Also concrete inconsistencies on code level can be avoided before code is available. Due to the

detailed specifications on analysis, design, or implementation level, components and their static or dynamic behavior are available much earlier in the development. Thus the components can be checked against each other for inconsistencies already on these levels, like described in the section 4.1.3(Static/Dynamic semantics of AUTOSAR interfaces) and section 4.1.5(Static/Dynamic semantics of EAST-ADL2 interfaces). Because individual inconsistencies can be recognized on an earlier stage of development, there are not as many inconsistencies to solve when components have to be integrated on code level.

Table 4.5 as well as appendix D summarize these results by providing a distinction between AUTOSAR and EAST-ADL2 and the part, which is contributed by the respective standard to enhance the individual basic conditions or to avoid inconsistencies. The table shows, except for a few exceptions, that EAST-ADL2 and AUTOSAR provide means to avoid reasons for inconsistencies as well as inconsistencies themselves in equal shares. However, individual drawbacks still remain.

In order to clarify the meaning of individual components and model elements on meta level M1, EAST-ADL2 as well as AUTOSAR just provides textual descriptions, which can be annotated with the respective element. Thereby it is indeed possible to support other developers' understanding of such an element, but due to the non-standardized textual character misunderstandings, misinterpretations, or missing information can not be avoided. Furthermore, even if developers are able to understand native text, machines are not. Because native text mostly contains implicit knowledge about components, which may be necessary for integration too, such information get lost presently for automating. For this reason the following section will introduce a possibility to standardize such textual descriptions in order to avoid misunderstandings and to enable using this kind of information also for automating.

4.2. Extension of AUTOSAR and EAST-ADL2 by Semantics

The last sections have identified three main points, which are not or only insufficient supported by language elements and specifications of AUTOSAR and EAST-ADL2:

- A missing standardized common development process or methodology, which prescribes amongst others roles and responsibilities
- Missing language elements for detailing the timing behavior or other dynamics of software components

	Enhancement from AUTOSAR	Enhancement from EAST-ADL2	Supports avoidance of:
Enhanced Basic Conditions			
Completeness & Clearness:	3	3	Misunderstandings, Inconsistent Processes
Consistency:	3	0	Loss of Information
Continuity&Traceability:	2	2	Missing Information, Mis- understandings, Lacking Maintenance
Integration of all disciplines:	2	2	Misunderstandings
Maintenance of artifacts	1	1	Lacking maintenance of artifacts
Assignment of roles and re- sponsibilities	0	0	
Avoidance of concrete Inconsis- tencies			
Static semantics of interfaces	3	0	concrete inconsistencies: see table 4.2
Dynamic semantics of inter- faces	2	1	concrete inconsistencies: see table 4.3 and table 4.4

Table 4.5.: Benefits from EAST-ADL2 and AUTOSAR:(0=insufficient; 1=partly; 2=sufficient; 3=good)

- Missing language elements for detailing the semantical meaning and correlation of concepts in a standardized manner, especially on meta level M1

But as already mentioned, currently there is one project, which develops a development process for integration of basic software modules (compare to section 2.3.4), while another project called TIMMO (compare to section 3.1.2), standardizes a language for describing the timing behavior of software components in more detail. For this reason and in order to expect the results and experiences of these two projects, the following section presents an approach for detailing semantical expressiveness of AUTOSAR and EAST-ADL2.

Therefore section 4.2.1 and 4.2.2 will explain the importance of semantics and what semantics mean in this context. Afterwards, section 4.2.3 describes language elements for describing the semantics by the means of a meta model, before section 4.2.4 sketches a concrete application of that meta model and the benefits, which come along with it.

4.2.1. Separation UnSCom from AUTOSAR and EAST-ADL2

	Preface (administrative, technical, organizational, and general information) * release information/ baseline/ configuration, developer contact, concerned vehicle states,...		
	Classing * feature oriented component classification		
	functionality/ technical terms - Conceptual Interface Model (functional)	architecture/ configuration (logical)	implementation/ quality (physical)
static view (Structure)	Information Objects (data) * Semantics of data	Signatures * interfaces, operations, data types	Usability, Maintainability, Portability
operational view (Effects)	Functions (actions) * Semantics of functions	Assertions * Pre- & Post- Conditions, Invariants	Functionality
dynamic view (Interactions)	Processes (workflows) * Semantics of workflows	Order * Interaction Protocols, states, transitions	Performance, Reliability, Efficiency
Component			

Figure 4.35.: UnSCom Component Description: Overview

In comparison with EAST-ADL2 and AUTOSAR, also the UnSCom approach [50] (another component description framework which was described in section 2.5.2) provides a meta model to describe different aspects of (software) components. Thereby UnSCom provides five different levels (or pages, a term borrowed from UDDI [47]), which are called the white pages, yellow pages, blue pages, green pages, and gray pages. These levels can be used for describing components on different views and abstraction levels, which are subdivided into a total of eleven individual concerns,

as depicted in figure 4.35. However, unlike EAST-ADL2 and AUTOSAR, UnSCom does not intend to provide a meta model for describing components of the automotive domain. Instead UnSComs' meta model is geared to the description of services or application software components for business processes. Nevertheless, even if some language elements and attributes differ between these two domains, also UnSCom has to describe components, so they are able to be integrated into an entire architecture. Therefore UnSCom should describe concerns, which are necessary for the understanding of components as well as for their integration, similar to AUTOSAR and EAST-ADL2 and vice versa. For this reason the following listing will contrast concerns of UnSCom with concerns, which can be described by AUTOSAR and EAST-ADL2. This brief comparison will show whether both sides provide similar information to enable integration or not in order to get an evidence for the completeness of AUTOSAR and EAST-ADL2.

- **Preface(white pages):** White pages are used within UnSCom to describe general administrative information of components. For this reason the UnSCom meta model provides language elements to specify release, baseline and configuration information of components as well as contact information, which can be used for communication between developers, which are responsible for the respective component.

However, as described in the sections above, even such information are also provided by, for example, the AUTOSAR::AdminData package (see figure 4.8) and/or the EAST-ADL2::Support package (see figure 4.29)

- **Classing (yellow pages):** The yellow pages of UnSCom can be used for a feature oriented classification of components. This classification is used to describe what kind of features a component represents. Thereby it will be possible to exchange components, which are realizing the same or similar features.

This kind of classification is also provided by the EAST-ADL2 VehicleFeatureModel (VFM), which was mentioned in section 4.1.4.

By then, only general component information are given. In comparison with the sections above, these kinds of information can be used for enhancing individual basic conditions of development. By contrast the following will show some more detailed component information, which are provided by UnSCom in order to describe components on three different abstraction levels (functional, logical, and physical), which are split into three kinds of views (static, operational, and dynamic). This fragmentation is also depicted in figure 4.35.

- **Physical level (gray pages):** UnSComs' gray pages are used to describe qualitative properties of components. Thereby individual requirements concerning the static view of components (e.g. usability, portability, and maintainability) are described. Furthermore, while an operational view is used to detail the components' functionality by some requirements, the dynamic view describes requirements, like performance, reliability, and efficiency of components, in order to clarify the components' dynamic behavior.

On the other hand, the sections 4.1.4 and 4.1.5 have shown, that EAST-ADL2 provides language elements to describe such requirements too. AUTOSAR also provides some requirements by the means of implementation descriptions, like mentioned in section 4.1.2 (see figure 4.3).

- **Logical level (green pages):** The green pages of UnSCom abstract the respective views of the physical level, which was described in the last bullet, in order to describe components on a logical level. This level specifies operation, interfaces, and data types of a component within its static view. Beside that the operational view can be used to specify assertions, like pre-, postconditions or invariants for operations and interfaces, which are specified within the static view. Furthermore the dynamic view provides means to specify interaction protocols, by which components are able to communicate. For this reason the dynamic view on logical level provides states and transitions in order to define some kind of deterministic finite automate, which prescribes allowed interaction sequences between components.

These kinds of information are also provided by EAST-ADL2 in combination with AUTOSAR. While AUTOSAR provides more language elements for describing operation, interfaces, and data types like described in section 4.1.3 and in section 4.1.3, EAST-ADL2 provides possibilities to describe pre-, postconditions or invariants, by the means of triggerConditions and OCL, as described in section 4.1.5. In addition, these triggerConditions as well as AUTOSARs' ModeDeclarations, which were also discussed in section 4.1.3, can be used for describing allowed transitions between different component states. This means, that AUTOSAR and EAST-ADL2 are able to define some kind of interaction protocol too.

- **Functional level (blue pages):** The blue pages abstract the logical level of UnSCom. On this level, UnSCom provides language elements to specify a system of concepts, which clarifies the semantics of concepts as well as their relationships to other concepts. These concepts concern all information objects (data), functions, and processes, which are necessary for data

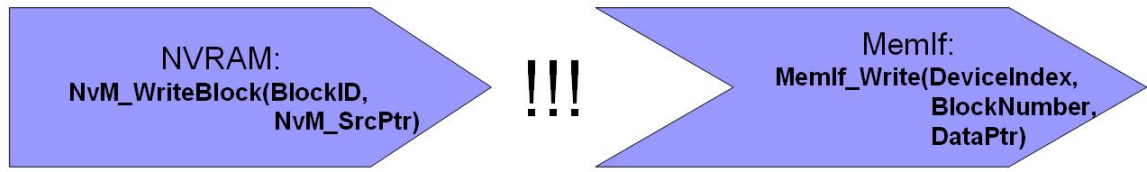
processing realized by the logical and physical level of UnSCom. To accomplish that, UnSComs' functional level provides a meta model too. This meta model is able to replace textual descriptions in form of native text, which normally are used to describe the concepts and meanings of elements, by a predefined meta model, which is able to represent an adapted form of native text in a standardized manner. Due to the standardized character of the meta model, these kinds of information also can be used and interpreted automatically.

By contrast, EAST-ADL2 and AUTOSAR describe such concepts from a software architecture point of view exclusively. Other aspects, which concern the conceptual understanding of data, functions, or processes or other additional information on model level M1, are annotated in form of native text, which cannot be interpreted automatically.

This listing has shown, that EAST-ADL2 and AUTOSAR provide almost the same information and abstraction levels like another component description framework called UnSCom. AUTOSAR and EAST-ADL2 furthermore provide additional standard specifications or guide lines, which can be used for enhancing the basic conditions of development more than with UnSCom. On the other hand UnSComs' functional level provides information, which are not provided by any language element of EAST-ADL2 or AUTOSAR. This particularly concerns the textual descriptions, which are used by EAST-ADL2 and AUTOSAR to detail semantics and conceptual properties of an model element on meta level M1, as described in section 4.1.6. Therefore the missing of such a description technique, which is provided by UnSCom, has been pegged as a critical drawback.

In order to get a better understanding for this semantical drawback and in order to show the limitations of AUTOSAR and EAST-ADL2, section 4.2.2 exemplifies a situation, where neither AUTOSAR nor EAST-ADL2 are able to describe necessary information in a standardized manner. Afterward, section 4.2.3 introduces a meta model of concepts, which is able to complete EAST-ADL2 and AUTOSAR by standardizing these semantical information. By the means of that meta model section 4.2.4 demonstrates how to complement the example of section 4.2.2 with missing semantical information.

Technical Level:



Semantical Level:

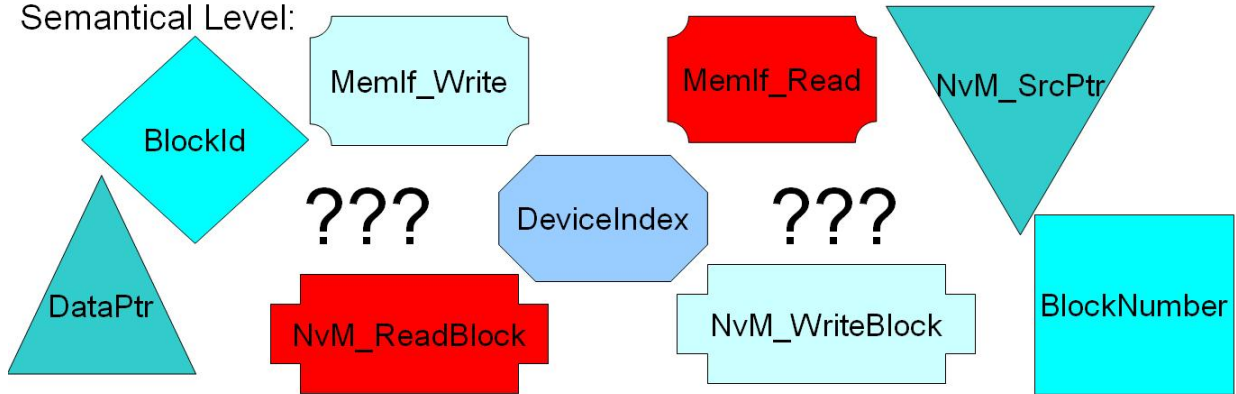


Figure 4.36.: Semantical Problem (schematic)

4.2.2. Demonstration and Concretion of AUTOSARs' and EAST-ADL2s' semantical drawback

In order to demonstrate the semantical drawback of AUTOSAR and EAST-ADL2, an example was taken from the memory management of AUTOSAR (compare to figure 4.14). First of all figure 4.36 shows two levels. A technical level, which stands for all existing AUTOSAR descriptions, and a semantical level. The technical level depicts the NVRAM Manager and the Memory Abstraction Interface in the context of a "Write" operation, which copies data from one memory block (the RAM block) to another memory block (the NV block). Therefore the NVRAM manager, which provides other modules with the `NvM_WriteBlock` operation, calls the `MemIf_Write` operation of the Memory Abstraction Interface, which abstracts from different underlying memory drivers and storage medias as depicted in figure 4.17. However, AUTOSAR specifications of the NVRAM Manager [24] and the Memory Abstraction Interface [23] only prescribe APIs and individual dynamics of `NvM_WriteBlock` and `MemIf_Write` by the means of the AUTOSAR meta model and additional sequence diagrams. By contrast, many semantical information and correlations between these two operations and their parameters are expressed in form of native text distributed across multiple documents.

For integration it is important to understand that `NvM_WriteBlock` needs two parameters, which are

passed to MemIf_Write. A NvM_SrcPtr, which identifies the RAM block from which the data are read from, and a BlockId, which identifies a NV block to which the data are written. Furthermore, the Memory Abstraction Interface operation MemIf_Write basically provides the same functionality, but the operation itself and its parameters are renamed according to Memory Abstraction Interface conventions. For this reason it is not only necessary to understand the sequence of calls, but also the correlation between parameters of the NVM (NvM_SrcPtr, BlockId) and the MemIf (DeviceIndex, BlockNumber, DataPtr). There it is indeed possible to check whether data types of individual parameters between two operation calls match, but it must also be ensured that operations use these parameters correctly. However these kinds of information are not expressed by the means of AUTOSAR or EAST-ADL2 in a standardized manner. Both standards are not able to express for what individual concepts are used for and how, without using textual descriptions.

This concerns the semantical level, which is depicted in figure 4.36. On this level, firstly there are multiple disconnected concepts, which are in relationship with other concepts anyway. Human integrators indeed are able to read multiple specifications in order to understand the conceptual interplay between the two components NVRAM manager and Memory Abstraction Interface. While machines are not able to interpret textual information for an automated integration at all, human integrators may also misinterpret them, whereby inconsistencies are preassigned like mentioned in the sections above.

Especially if multiple operations provide similar or equal signatures, it could be difficult to decide which operations should be connected.

For example: Figure 4.36 also shows two additional operations of the memory management on the semantical level. While NvM_ReadBlock operation and the MemIf_Read operation specify almost the same signatures, like NvM_WriteBlock and MemIf_Write, in terms of their parameters, here the parameters are used for a “Read” operation. An integrator may possess implicit knowledge, which enables him to decide, that Write and Read operations have to be handled variably. However, generally based on simple syntactical information, which is given by the AUTOSAR and EAST-ADL2 meta model, it would be possible to interconnect a “Write” operation of the NVRAM manager with a “Read” operation of the Memory Abstraction Interface.

This brief example of the AUTOSAR memory management sketches some problems concerning the integration of Basic Software components, like described in section 2.3.2. Because each AUTOSAR Basic software module is described by its own standardized specification completely and exactly, it does not seem to be necessary to standardize the textual descriptions further more. Unfortunately,

standardized specifications only are available for basic software and not for application software. Therefore developers may introduce their own concepts, which are not standardized and described inside of common AUTOSAR documents like for basic software modules. Especially relationships between multiple concepts and correlations among similar concepts, which can be used for reuse or exchange of concepts between different components, remain obscurely.

In order to describe such information in a standardized manner, the following section 4.2.3 will show a meta model for describing conceptual relationships, before section 4.2.4 exemplifies its usage to express the above mentioned semantical information.

4.2.3. Linguistic Meta Model and Benefits

Generally, the new approach will provide means to describe semantics and conceptual properties for model elements on meta level M1 without using the native text. For this reason a semi formal way, which is realized by the means of a meta model too, was chosen. This meta model either can be integrated into AUTOSAR and/or EAST-ADL2 by the means of a meta model extension of the respective meta model, or via references. Such references would reference from AUTOSAR or EAST-ADL2 language elements to the external description like realized, for example, by an EAST-ADL2 ExternalBehavior (see section 4.1.5 or figure 2.2.2).

However, this thesis primarily does not purpose to prescribe how and whether the new approach will be integrated into the existing standards. Instead, the following section describes the language elements, which can be used to specify the desired information, and their usage as well. In order not to reinvent the wheel and because the functional level of UnSCom even provides a meta model for the desired concerns, the new approach of this thesis will (re-)use an adapted version of UnSComs' functional level meta model.

Meta model for describing concepts and semantics: Overview

An overview on the new meta model and its language elements is depicted in figure 4.37. There the central class, which has to be detailed by other language elements, is the component. A component could be realized, for example, by a software component template (compare to section 2.1.2), an EAST-ADL2 entity or any other component description. The figure shows, that a component associates a lexicon as some kind of terminology, which aggregates all concepts, which are used for this component. Concepts can either be information objects, functions, or processes. While information

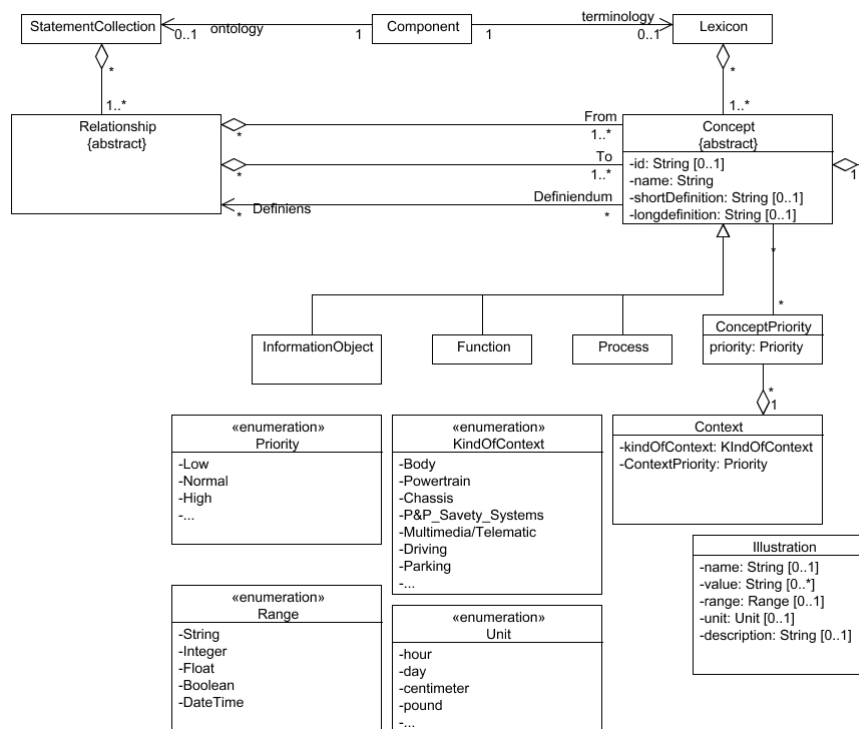


Figure 4.37.: New Concepts: Overview

objects are describing things in general, functions describe activities, which transform the information objects, and processes describe sequences of multiple functions.

Beside the lexicon, a component also defines a `StatementCollection`, which is used to relate a concept to other concepts of the lexicon. This is realized based on the meta model, which prescribes allowed relationships between information objects (see section 4.2.3), functions (see section 4.2.3), and processes (see section 4.2.3).

After defining all component concepts, it is possible to detail the meaning of a concept itself by the means of some application-independent statements. For this reason the abstract class `Concept` primarily provides `short-` and `longDefinition` attributes in the form of native text. Additionally these two attributes can also be expressed by the means of the standardized meta model grammar via the “Definiens-Definiendum” association. Thereby the concept is related to other component(-independent) concepts via relationships like above. This means that it is possible to replace or complement native text as well. Therefore, according to native text, an information object can be seen

as substantive, while functions are used to describe verbs in a linguistic context. Like described in section 4.2.3, information objects and functions can also be used to express statements in a familiar linguistic word order (Subject (information object) - Predicate (function) - Object (information object)) by the means of the meta model. After that, processes can be used to bring functions or statements into a chronological ordered sequence.

For example, a system of concepts may be composed of the following concepts: Five information objects called "Pointer", "NvM_SrcPtr", "DataBufferPtr", "Data", and "RAMData" as well as one Function called "point to". By the means of these concepts, it is now possible to define statements via relationships like:

- «A "Pointer" can be a "NvM_SrcPtr" or a "DataBufferPtr".»
- «A "Pointer" "points to" "Data".»
- «"Data" can be "RAMData".»

After that, it is possible to combine all statements, that the following complex statement can be expressed:

- «A "Pointer", which can be a "NvM_SrcPtr" or a "DataBufferPtr", "points to" "Data", which can be "RAMData".»

Furthermore, a concept can also define some priorities, whereas a particular priority depends on the context within the concept used. For this reason, an individual context, which also defines an priority dependent on the application field of a context, defines some priorities in order to aggregate concepts of the same priority within one certain context. These kind of information can be used for runtime optimizations in two ways. Firstly, contexts and priorities can be used to differ between functions in order to indicate whether a function is relevant within an individual context or not.

For example: a specialized function "read Data" is defined in the context of "P&P_Safety_Systems", while another function "read Data" is defined in the context of "Body". As safety has a higher priority than body definitely, the operating system can decide for which kind of function it will provide more resources.

Secondly, within the same context it is also possible to differ between priorities.

For example: the two function "read Data" and "write Data" could possess the same context. However, by the means of additional concept priority information, developers can define that "read Data" must have a higher priority than "write Data". This means, that an operating system can also specify that

a task, which controls “read Data”, is prioritized more than a task, which controls “read Data”.

Meta model for Information Objects

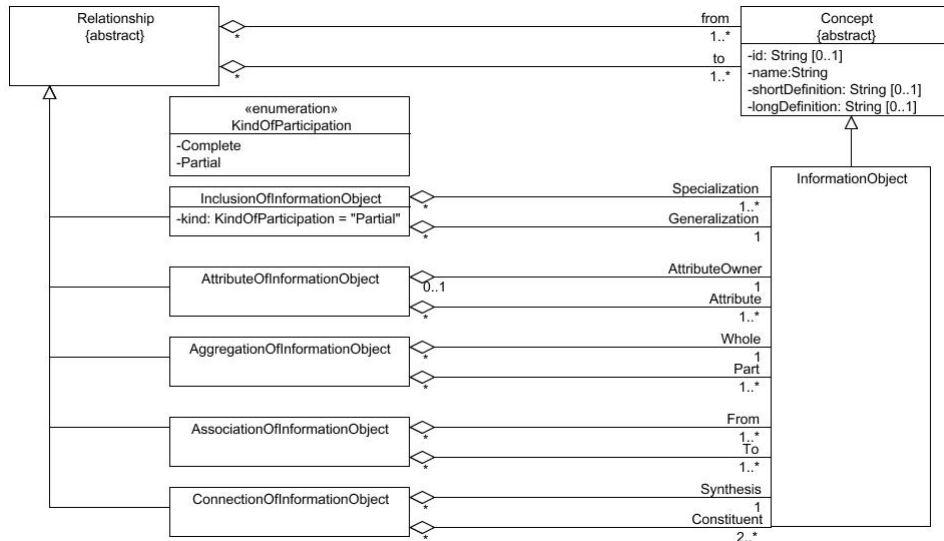


Figure 4.38.: New Concepts: InformationObject

Figure 4.38 depicts the concept of information object and allowed relationships between multiple information objects. The meta model defines five different relationships to clarify correlations between information objects. The following listing will detail the meaning of these relationships and differences between them.

- InclusionOfInformationObject:** Conceptual hierarchies between information objects are represented by inclusion relationships. Thereby an inclusion assigns at least one specialized information object to exactly one general information object. The specializations can be seen as general information objects with additional more detailed information, which are interconnected via the logical operation “XOR”(exclusive or).

Example:

«A “logical block number” can be a “BlockNumber” or a “BlockId”».

Furthermore, the kind attribute of the inclusion relationship indicates whether the relationships builds a partition or not. In the case that an partition is build, the “kind” attribute, whose default is “Partial”, is “Complete”.

Example:

While the sentence «A “MemoryDriver” can be a “FlashDriver” or a “EEPROM Driver”.» does not build a partition (the kind attribut is “Partial”), the sentence «A “logical block number” can be a “BlockNumber” or a “BlockId”». build a partition (the kind attribut is “Complete”).

- **AttributeOfInformationObject:** Attribute relationships describe essential constituent properties of information objects. The attributes are used to describe individual properties of their AttributeOwner.

Example: «A “MemoryDriver” has a “DeviceIndex” and “Name”.»

In the case that an information object is specified by more than one attribute, the attributes are linked by a logical “AND”. In addition, by the means of the Illustration meta class (see figure 4.37) it is further possible to define individual parameter value for an attribute or concepts in general. For example: “A date is a DateTime measured in days” or “A name is a String”.

- **AggregationOfInformationObject:** An Aggregation is used to express, that values of one or more concepts are an integral part of another concept. Unlike inclusion relationships, the part concepts can not be seen as special case of the Generalization. Unlike attributes, the part concepts are stand-alone concepts describing the properties of a concept.

Example:

«A “MemoryStack” is composed of a “Memoryservice”, a “Memory Hardware Abstraction”, and “Memory Driver”.

- **ConnectionOfInformationObject:** A special kind of Whole-Part relationships are concept relationships, whose relationships lead to new concepts. For this reason the relationship assigns at least two Constituents, which build together the Whole to a Synthesis.

Example:

«A “logical block number” is a correlation of “BlockIdentifier” and a “data index”.

- **AssociationOfInformationObject:** The AssociationOfInformationObject relationship, can be used if other relationships are not able to express the desired relation, or if relations between information object can not yet be clarified at a certain point of development. It is a form of compromise in order to represent basic relationships, which should not be used if other relationships are more suitable to express the relation.

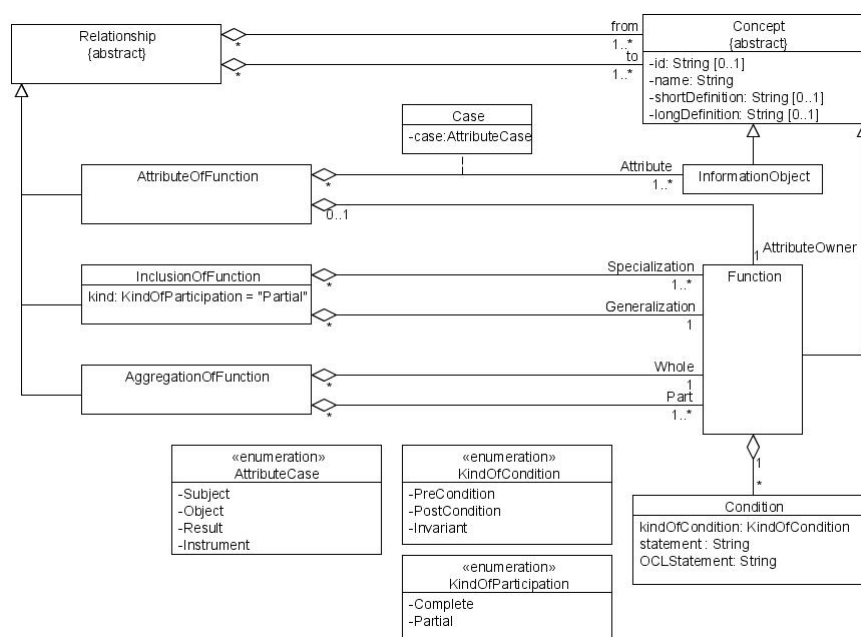


Figure 4.39.: New Concepts: Function

Meta model for Functions

Figure 4.39 depicts relationships, which can be used to refine functions by the means of aggregations and inclusions step by step. Beside these two relationships, the `AttributeOfFunction` relationship assigns information objects to functions in order to describe characteristic properties of the function.

- **InclusionOfFunction:** Like information objects, also functions can be specialized via InclusionOfFunction relationships. This can be used if functions can be summarized into a general class of functionality.

Example:

«MemIf_Write» executes either «EE_Write» or «Fee_Write». (This statement does not build a partition, because there are more functions, which can be used by the Memory Abstraction Interface to store data into other kinds of memories than Flash or EEPROM. For this reason the kind attribute has to be «Partial» in that case.)

- **AggregationOfFunction:** An AggregationOfFunctions aggregates functions, which are necessary to accomplish one single job. Like AggregationOfInformationObject, the aggregation here relates several Parts to one Whole.

Example: «A “Write” Function executes of “NvM WriteBlock” and “MemIf Write”»

- AttributeOfFunction:** Generally, there are three characteristic properties, which can be assigned to a function: The Subject, which accomplishes the function, the Object which is concerned by the function indirectly or directly, and the situation, which describes the context during execution by the means of states before and after execution. For this reason the meta model provides an additional “attributeCase” for attribute associations, which assign different properties/information objects to exactly one function according to their respective meaning in the context of that function. Thereby the attributeCase indicates whether an information object accomplishes the function (Subject), is used for the function directly (Object), supports the function indirectly (instrument), or result from the function (Result). The situation, within the function is executed, is described by conditions, which are aggregated by the function concept, in combination with the conceptContext. The conditions are used to describe pre-, postconditions, or invariants and can be described by the means of OCL or other logical formulas in form of Strings.

Example:

«The “NVRAM”(Subject) does “MemIf_Write”(Function) by the means of a “NvM_SrcPtr”(Instrument), a “DeviceIndex”(Instrument), and a “BlockId”(Instrument).» or «A “Pointer”(Subject) “points to”(Function) “Data”(Object).»

Meta model for Processes

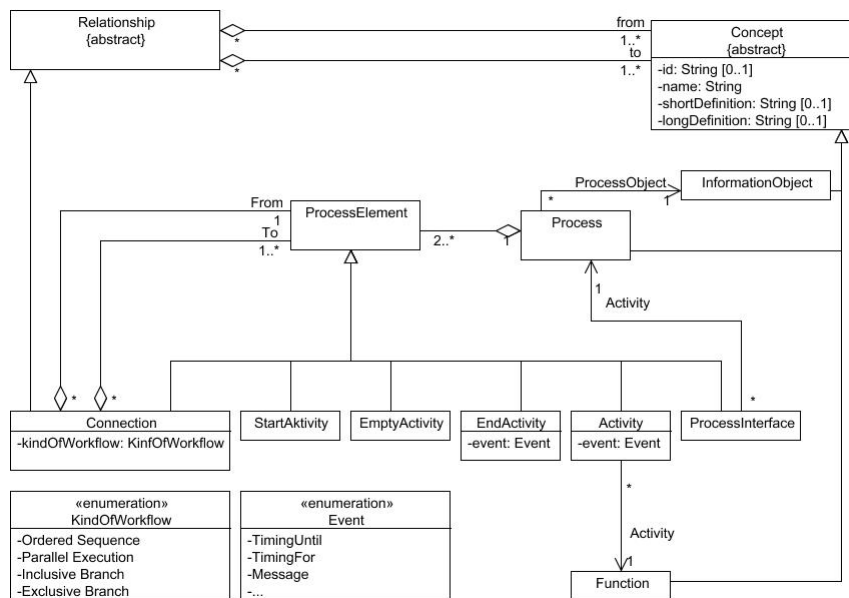


Figure 4.40.: New Concepts: Process

Interactions, which define chronological ordered application flows and causal dependencies of functions (workflow), can be described by language elements, which are depicted in figure 4.40.

The central language element here is the Process. A process attends to exactly one ProcessObject by executing at least two ProcessElements. The two obligatory ProcessElements, which must be part of any process, are a StartActivity as entry point and an EndActivity as exit point of the process. Between entry and exit point there can be Activities, which are associated with exactly one function, process interfaces, which serve as link between two independently defined processes, and empty activities. Thereby activities have a triggering event, whose occurrence will execute the associated function. Such an event can be a TimingUntil event, which indicates that a function waits for a certain point in time, a TimingFor event, which indicates that a function waits for a certain period, or an Message event, which indicates that a function waits for the arrival of a certain message.

Additionally, in order to describe workflows process elements must be related to other process elements by the means of the Connection relationships, which can express one of the following four dependencies:

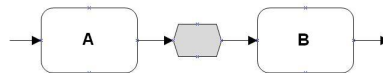


Figure 4.41.: Ordered Sequence

OrderedSequence: An OrderedSequence relationship, compare to figure 4.41, which indicates that an activity B must be executed after an activity A has finished.

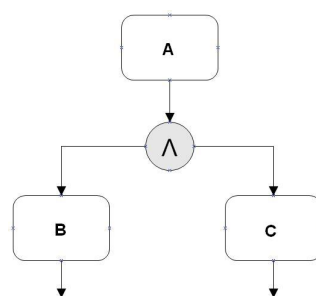


Figure 4.42.: Parallel Execution

ParallelExecution: An ParallelExecution relationship, compare to figure 4.42, which indicates that a parallel execution of activity B and activity C follows activity A.

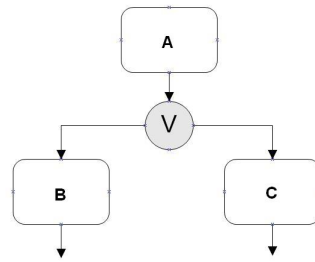


Figure 4.43.: Inclusive Branch

InclusiveBranch: An InclusiveBranch relationship, compare to figure 4.43, which indicates that either activity B or activity C or both activity can follow activity A.

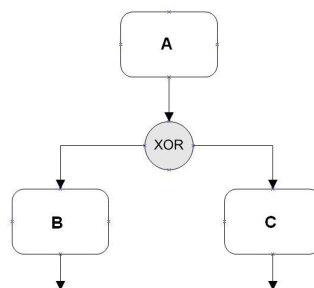


Figure 4.44.: Exclusive Branch

ExclusiveBranch: An ExclusiveBranch relationship, compare to figure 4.44, which indicates that either activity B or activity C must follow activity A.

By the means of such workflow descriptions it is possible to define complex correlations and chronological dependencies between functions. This should be exemplified by an example, which is taken from the specification of the AUTOSAR NVRAM manager [24]. A sequence diagram, which is also shown in figure 4.23, defines the following statement, which can be expressed now by the means of the new meta model:

Example:

«The Process “Write” deals with “RAMData” and executes “NvM_Write” then “MemIf_Write” then either “EE_Write” or “Fee_Write”.»

4.2.4. Application & Case Study

Appendix E shows a continuous example in order to demonstrate the usage of the aforementioned meta model (see section 4.2.3). First of all E.1 shows a continuous textual descriptions of statements, which are used to describe the functionality of “Write” in the context of the AUTOSAR Memory Stack. These statements extend the statements of the latter section in order to represent the missing information by standardizing the semantical conceptual relationships, which were described in section 4.2.2. These continuous statements are then subdivided into single statements from which the whole statement can be derived. After these textual representations, appendix E.2 depicts a graphical representation of these statements modeled by the means of the meta model. All concepts and relationships among them are represented in graphical form by the last figure in section E.2.

Benefits from the Linguistic Meta Model

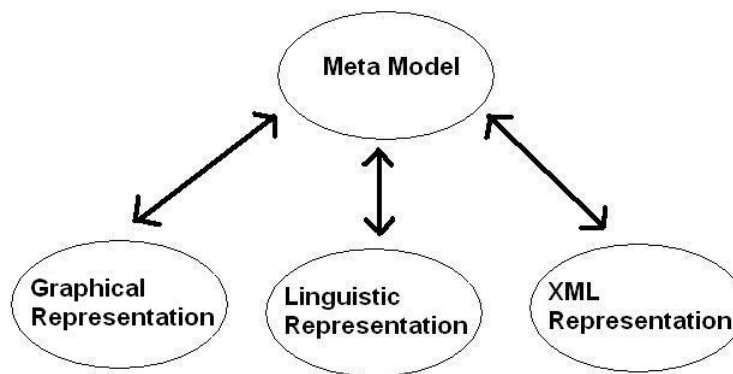


Figure 4.45.: Transformation between multiple representation format, following [50]

Semantical information are not only annotated in form of key, value pairs like, for example, within the CCI approach, which was described in section 2.5.1. Instead, the aforementioned meta model is able to represent relationships between concepts in order to clarify their semantical meaning without using a predefined set of information or values. Like for linguistic text, correlations are described via relationships to other concepts, which enables to detail the semantics more than by the means of CCI or other current standards.

Furthermore, information are not only presentable in form of UML models like depicted in appendix E.2. As sketched in figure 4.45, it is also possible to define automatic transformations, which con-

vert these models into linguistic text, a XML exchange format, or vice versa. This enables each developers to choose a representation format with which he/she is more familiar, which enhances understanding further more.

By the means of additional tool support, developers can be supported to reuse or to extend existing concepts. All predefined concepts can be used to describe their own semantical information by combing these concepts via standardized relationships. After defining multiple statement collection or lexicons for each respective system component the lexicons are able to be integrated or merged via the aforementioned exchange format XML. Thereby it is possible to find redundant information objects, functions, or processes, by what similar data , operations, or operation sequences are able to be recognized, reused, or optimized.

Beside these benefits, the additional meta model also supports the avoidance of inconsistencies from section 2.4.2, like described in the following:

- **Syntactic inconsistencies:** The semantical meta model provides a basis for data (information objects) and operations (functions), which are (re-)used and detailed on the abstraction levels of EAST-ADL2 and AUTOSAR by technical information. As concepts do not only represent semantical information, but also the static relation between functions and parameters implicitly, concepts can also be used to avoid syntactical inconsistencies concerning ports, interface, signatures and data names on an early stage of development.
- **Semantical inconsistencies:** Basically, the additional meta model comes along with the avoidance of any linguistic ambiguities of data (names), operations, and processes. This is the ultimate advantage of this meta model. However, by the means of the Illustration class, which details the data type semantics of concepts, also other semantical inconsistencies concerning, for example, numerical or reference system properties can be avoided on an early conceptual stage of development.
- **Application-based inconsistencies:** By the means of the description of Functions and Processes, also sequences of operations can be specified, by which state violations can be avoided. Especially the description of Functions in combination with their Attributes avoids inconsistencies concerning the correct usage and order of parameters for a clearly defined task.
- **Pragmatic inconsistencies:** Generally, the meta model also provides language elements to describe statements, which can be used to avoid pragmatistical inconsistencies. But for such ad-

vantages other approaches like AUTOSAR and EAST-ADL2 themselves or TIMMO are more suitable than this meta model.

All in all, the meta model comes along with a better understanding for developers concerning semantical relationships of concepts on meta level M1. This does not only support integrators, but it also the avoidance of inconsistencies. Furthermore, because semantical information will become machine-readable, these information can be used for an automated integration of system components too.

Chapter 5.

Conclusion and Outlook

5.1. Summary

The last chapters have analyzed a way to enhance integration for automotive software components into an entire architecture. Therefore chapter 2 introduces a standard called AUTOSAR and an architecture description language called EAST-ADL2. These two means intend to enable a standardized description of automotive architectures on a model level across several abstraction levels and concerns. To lift integration up to an early stage of development section 2.3 has described integration, i.e. integration of model and views in contrast to an integration of implemented modules.

As the thesis' goal was to solve inconsistencies during integration, section 2.4 introduces inconsistencies generally, before a survey was made in order to concretize current inconsistencies in practice without AUTOSAR or EAST-ADL2. The survey, which is described in section 3, has analyzed more general problems concerning the working environment of the interviewees, which are hampering integration through bad basic conditions of development more than mismatching software components themselves. However, concrete inconsistencies problems concerning the integration of software components were analyzed as well. Thereby the survey has shown, that concrete inconsistencies are hampering integration less predominantly than mismatching basic condition of development.

For this reason, chapter 4 has analyzed how far the basic conditions of development can be enhanced and how far concrete inconsistencies between software components can be avoided by the means of AUTOSAR and EAST-ADL2. That analysis found out, that AUTOSAR as well as EAST-ADL provide a complete and clear meta model, which is able to specify models of software components

completely and clearly according to their respective concerns. While AUTOSAR supports consistency of artifacts more than EAST-ADL2, both specification together are able to integrate various disciplines and stakeholders. Therefore they also provide good means for tracing among all artifacts, models and views continuously in order to ensure traceability of individual solutions across multiple development levels. And in order to standardize a common maintenance of artifacts for their exchange, both provide similar concepts, which are helpful but upgradeable. However, both specifications neglect an agreement concerning a common development process or methodology for prescribing how to use the respective specification. Especially roles and responsibilities are not defined, so that development delays are caused by a different alignment of several development processes distributed across multiple enterprises, divisions, and developers.

Beside these basic conditions, the analysis has shown, that AUTOSAR in combination with EAST-ADL2 provide good support to avoid almost any inconsistency, which were identified in section 2.4. They are able to avoid syntactical and semantical inconsistencies almost completely. While the most application-based inconsistencies can also be avoided, some other dynamic inconsistencies concerning the timing behavior and state transitions of components can not be described sufficiently, by what some integration faults still remain.

All in all the analysis has shown, that AUTOSAR in combination with EAST-ADL2 provide good support for enhancement of basic conditions and the avoidance of inconsistencies in equal shares. But by contrast, misunderstandings of concepts on model instance level M1 represent a critical problem and may lead to any kind of inconsistency. Because neither AUTOSAR nor EAST-ADL2 provide language elements to describe conceptual semantics without using native text and because there are already two current projects for enhancing the problems concerning the development process and the timing behavior of components, section 4.2 has concentrated on semantical problems to avoid misunderstandings in general. Therefore this section has introduced a meta model for describing semantical concepts by predefined relationships. By the means of a short example the meta model was demonstrated in order to show benefits, which come along with these new semantical information.

5.2. Outlook

As this thesis has concentrated on software components in general, future research must analyze integration problems from a more detailed point of view. As described in section 2.3.2 AUTOSAR supports several components, which must be brought together. This does not only concern different software component types, like application and basic software, but also hardware and mechanical components. As each component has its own requirements on integration, these individual requirements also must be taken into account in the future.

Furthermore, the actual problems, which were identified by this thesis, must be solved by further development of AUTOSAR and EAST-ADL2. This particularly concerns the development of a process or process framework, which recommends a development process, which can be applied by any division to simplify their collaboration. Moreover, the TIMMO project indeed intends to enhance dynamics and timing behavior of software components, but because dynamics is a wide area, it should be analyzed further more.

However, not only critical problems must be enhanced by further research. Also the points, where AUTOSAR or EAST-ADL2 just provide partial or just sufficient support, must be enhanced or extended by, for example, guide lines or best practices. Especially EAST-ADL2, which basically provides just its meta model, must be extended by further specifications like AUTOSAR. Because AUTOSAR specifies not only a meta model, but also, for example, tool interoperability, modeling rules, conformance, and model persistence rules, such specifications should be entered in EAST-ADL2 too.

Beside these objectives the semantical support must be further developed. Perhaps by the means of meta model extensions or additional support in form ontologies, EAST-ADL2 and AUTOSAR concepts and their semantics should be defined clearly in order to enable an automated integration beside just syntactical or statical information. For this reason, tools have to be developed. These tools must support developers in handling of AUTOSAR and EAST-ADL2 as well as in defining their semantical information. Furthermore, tools or rather the meta models should enable the specification of tests. These tests should be able to be generated from models automatically, in order to check or to simulate functionality and interoperability of components already on model level further more. Such tests could be exchanged between divisions like models, in order to check individual scenarios between foreign components and the own.

There is a lot of work planned in the future of AUTOSAR and EAST-AD2, but if it works, annoying work can be avoided even more than already today in order to concentrate on functionality and

quality of automotive systems.

Appendix A.

Acronyms

ADL Architecture Description Language

ASW Application Software

ATESST Advancing Traffic Efficiency and Safety through Software Technology

AUTOSAR Automotive Open System Architecture

BSW Basic Software

CCI Consistent Component Integration

ECU Electronic Control Unit

E/E electric/electronic

FAA Functional Analysis Architecture

FDA Functional Design Architecture

MDA Model Driven Architecture

MDD Model Driven Development

MDE Model Driven Engineering

MDSD Model Driven Software Development

NV Non Volatile

NVM Non Volatile RAM Manager

OEM Original Equipment Manufacturer

OMG	Object Management Group
RAM	Random Access Memory
ROM	Read Only Memory
UDDI	Universal Description, Discovery and Integration
UML	Unified Modeling Language
UnSCom	(Unified Specification of Components)
RTE	Runtime Environment
SDL	Specification and Description Language
SWC	Software Component
SysML	Systems Modeling Language
VFB	Virtual Functional Bus
XML	Extensible Markup Language
XMI	XML Metadata Interchange

Appendix B.

Survey

Survey about problems of software integration in practice

Your Role and working environment

1. Please describe briefly your team and your function or your daily work within your team.
(I.e. what's your job title?)

2. With what kinds of developers of other disciplines do you have to work together regularly?
Do you have specific collaborations with other department/business units?
If yes, what other disciplines do they belong?

Communication

3. What means do you use for communication and agreements with other developers, i.e. if they are distributed over multiple locations?
(E.g. Phone, Mail, Meeting, Version Management Systems...)

4. During collaboration with other developers specifications and some other documenting artifacts have to be exchanged. How are these data represented for exchange?
How intensely are these individual representations used? Please try to give percentage estimate.

By models (e.g. UML)

 %

By textual descriptions

 %

By code %

By others % which

ones?

5. What kinds of problems do occur, when you have to collaborate with other developers?
(I.e. with developers with other expertise or with other cultural background)

6. What kinds of problems do occur, when you are working just with artifacts like specifications or models of other developers and without a personal contact?

Tool-Support

7. Which development tools do you use for your daily work? For what do you use these tools?

used

for:

used

for:

used

for:

used
for:
 used
for:

8. Are you supported by your Tools at the best? If you are thinking your tool does not support you well: can you give a short description of the problems you have with them?

9. In the majority of cases multiple Tools are in use for the same job or for different jobs.
These different tools may be used by you or by other developers you work with.
Is the tool landscape standardized?



Yes



No

Would a standard be helpful?



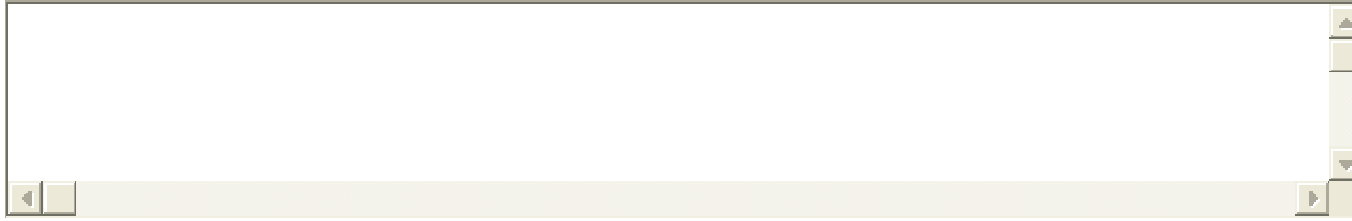
Yes, it would avoid some problems



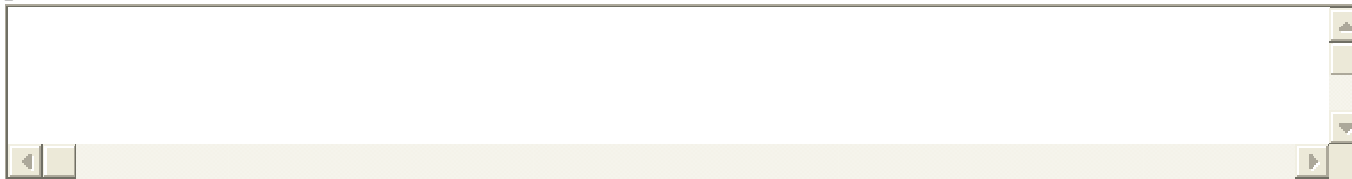
No, because this would raise more problems like:

What kinds of problems do occur mostly when different Tools are in use for working together?

10. Is an integration of artifacts produced by other developers into your tools or platforms always possible? What kinds of integration faults do occur?

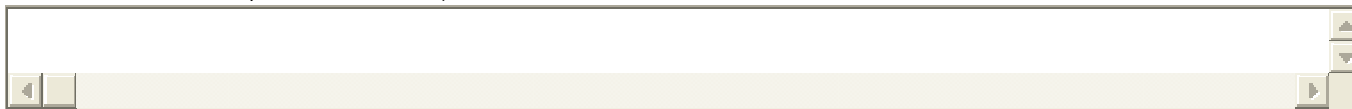


If you are working with models, are there any special kinds of problems?

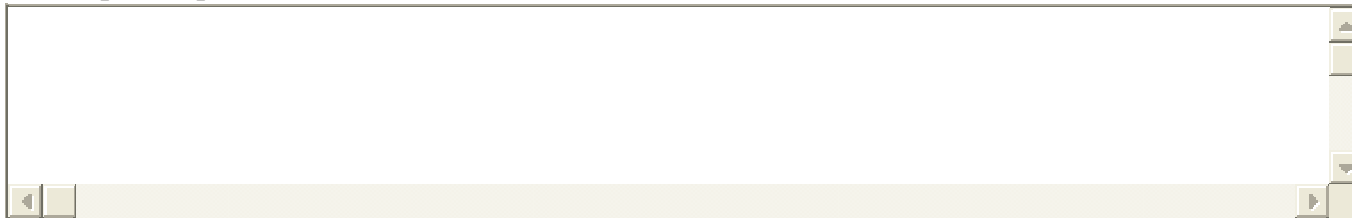


Development Process

11. What kind of development process do you apply? (Unified Process, Waterfall model, V-model...)



Can you give some examples of problems which do occur in terms of the development process and its artifacts?



Integration of different components into a whole


12. At which point inside of your development process plays integration with components produced by others (companies, teams, developers...) a relevant role for the first time?


What kinds of artifacts are there relevant?
(E.g. Textual descriptions or models or implemented modules...)

13. What problems do occur when you try to integrate several components of different developers into a whole?

14. What are possible reasons for such an integration fault?

15. Do you generally think integration of components on model level would avoid some problems?

 Yes, because

 No, because

The common survey is over at this point.

By now I would like to thank you for your contribution.

The following section is an optional part of this survey. It's not necessary to fulfill this part but it would be a big support if you are all set to answer these questions too.

Otherwise please **save** this document and **mail** it to the following e-mail address: survey@ds-lab.org

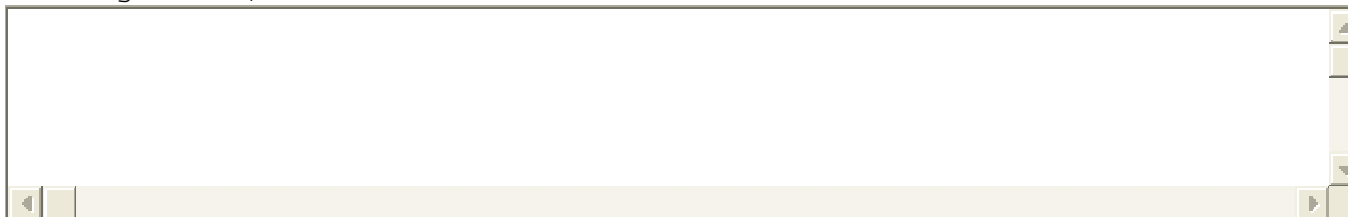
Many Thanks!

This part tries to find out more closely where problems avoiding a later integration of distributed developed components do arise. Please have a look at the questions and try to answer them if you are thinking that they concern your work.
You can end the survey anytime. Just save the document and mail it to survey@ds-lab.org

Artifacts before implementation

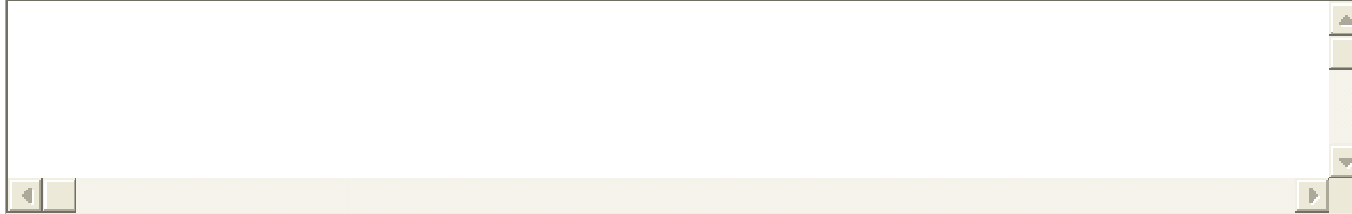
During analysis and design of a system many artifacts on different abstraction levels are created to refine a system from (non-)functional view down to a technical view. Are there any problems with management or understanding of such artifacts?

(Possible problems are: common inconsistencies regarding naming or syntax among documents/misunderstood artifacts caused by different knowledge bases)



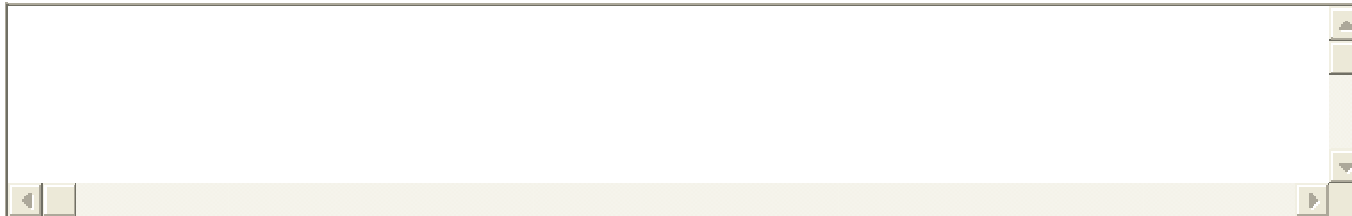
Specification vs. Implementation

It happens that programmers implement beside the specification. Do special problems compared to the former question arise from such non-compliance?

A large, empty rectangular text input box with a light beige background. It features a vertical scrollbar on the right side and a horizontal scrollbar at the bottom, both with small triangular buttons at their ends.

Semantic inconsistencies between components

Inconsistencies of semantic nature may occur, if data with different or ambiguous semantic is exchanged between components. (Examples are: synonyms, differing data ranges or data formats...)
What kinds of semantic inconsistencies do restrict you more often when you want to link your component to others? Can you describe these inconsistencies, please?

A large, empty rectangular text input box with a light beige background. It features a vertical scrollbar on the right side and a horizontal scrollbar at the bottom, both with small triangular buttons at their ends.

Application-based inconsistencies between components

Application-based inconsistency may occur if local functionalities of a component do not accurately reflect a global application context. This occurs if global constraints or constraints between components are not fulfilled by components.

(Examples are: violated state relations among components or mismatches between conditions of different components)

What kinds of application-based inconsistencies do restrict you more often when you want to link your component to others? Can you describe these inconsistencies, please?

A large, empty rectangular text input box with a light beige background. It features a vertical scrollbar on the right side and a horizontal scrollbar at the bottom, both with small triangular buttons at their ends.

Pragmatic inconsistencies between components

Pragmatic inconsistency concerns in particular concurrency constraints, timing requirements dedicated by hardware and algorithms as well as the underlying architecture, infrastructure or topology.

(Examples are: violated timing constraints among a sequence of operation calls or a faulty distributed access on shared variables)

What kind of pragmatic inconsistencies do restrict you when you want to link your component to others? Can you describe these inconsistencies, please?

Common questions

Do you have to apply any Guide Lines or Best Practices? Are they always followed by everyone?



Yes, they are hold



No

Are there any other reasons or problems concerning integration of components into a whole, which are not listed here?

Would you be prepared to participate a further survey dealing more detailed with integration problems?



Yes



No

Many Thanks for your contribution!

Please save the document and mail it to survey@ds-lab.org

Appendix C.

Analysis of the survey

Communication

- Used media:
 - Mail, Phone, Netmeeting, LiveMeeting, WebEx, regular meetings. Phone/video conference, project drive, version management systems , Databases, Wiki, WEB Portals
 - Knowledge exchange tool SharePoints, common data server
 - Project related data and documents like specifications: Document management systems like Documentum, Sharepoint, etc.
 - Software related documents and work outputs: Version management system “CM synergy / change synergy”
- Exchange Format:
 - Model: 9.5%
 - Text: 63.75%
 - Code: 19.25%
 - Others: 7.5%
- Problems:
 - Humans:
 - * Language problems (English is not the mother language for most, same wording is used for total different means).

- * Different cultural backgrounds are issues as well, and very different working style will come together
- Specifications:
 - * Incomplete/ unclear/ Contradictory documentation
 - * Different technical background / basic knowledge:
 - * Missing link to requirements (for what reasons a certain solution has been selected?)
 - * Missing link to other subsystems/components/disciplines (interrelationships between the artifacts)
- Process:
 - * Project schedules not detailed enough or not well communicated. Incomplete or missing project framework (project plans, work packages planning, project objectives, constraints and restrictions)
 - * incomplete or inhomogeneous distribution of information among departments and teams
 - * incomplete or missing risk management
 - * Different roles have different targets and interests/ Responsibilities not defined clearly.
 - * At former project a risk analysis has not been performed -> direct contact to the function developer of former projects

Tool-Support

- Are the Tools standardized? Yes: (1) No: (7)
- Would standardization help? Yes: (6) No: (2)
- Tools are generally o.k.? Yes: (4) No: (3) No explicit opinion: (1)
- Comments:
 - Tool support is fair but not excellent.
 - Too many different tools

- Standardization will limit innovation
- Used Tools:
 - Telelogic Synergy/ Telelogic Change used for: Configuration management/ Change management
 - Doors used for: Requirements engineering
 - Microsoft Office used for: Documentation:
 - Enterprise Architect used for: modeling
 - Matlab/Simulink used for: System simulation and function development
 - OmniBuild used for: for make in order to address a complicated build environment for a very large system
 - OSS Compiler Tool chains used for: Getting the stuff compiled/linked (VxWorks, Java, CORBA, etc)
 - CONAS used for: Calibration
 - MKS used for: CM
 - PVCS used for: Common CM tool
 - CM Synergy used for: SW-CM tool
 - Change Synergy used for: Common CR-tool
 - LIMAS used for: Software Specification tool
 - XD/Dataspy used for: Auxiliary function development tool to check interfaces of specification
- Problems:
 - Incompatible file structures
 - Incompatible headers in files
 - Incompleteness of the meta model
 - Inconsistencies, because single source principle is not applied
 - Information exchange with OEM is not standardized in a formal way, like doors eXchange

- Effect:
 - A lot of manual checking/ adaptation/configuration/transformation of artifacts for use with different tools is necessary/ Large training effort
 - A lot of experience is necessary to produce usable code from models
 - During export and import between Tools one always loses information: (at least partially or for some attributes/capabilities)
 - Different versions of tools/libraries/configurations being used.
 - Incomplete or contradictory specifications, particularly concerning the interfaces
 - Implementation errors not detected at the component test slow down system integration drastically
 - Incomplete requirements cause unintended system behavior and related integration faults
 - Data models of the different tools do not fit to each other.

Development Process

- Applied processes:
 - Variants of the V-model
 - RUP
- Problems:
 - Due to project schedule restrictions the documentation is neglected
 - Problems occur due to incomplete execution of the process and inappropriate restrictions during development work (project timing, resources, risk management etc.).
- Possible Solution:
 - An overall process defining all roles to be defined in an understandable and easy way for each step:
 - * What needs to be done?
 - * What is the responsibility of the role?

Integration of different components into a whole

Model Integration could be helpful?

- YES, because:
 - One gets an overview earlier and can address more precise test cases
 - An (HW/SW/mechanical) spanning view of the system can be applied, in order to define a system structure
- No, because:
 - The main issue is the time pressure for integrators.
 - Most of the problems are detected and handled on C-implementation level.
 - We need a new generation of modeling tools which can handle variant management and product lines much better than today's tools can do.
 - We need a new approach of describing the semantics of interfaces (data types are not sufficient, we need for example production rates, accuracy, detailed meaning, etc.)
- Involved Artifacts:
 - Implemented modules
 - Textual descriptions when testing against requirements is executed.
 - Interface descriptions
 - Models

Comment:

Changes have to be annotated by a textual description providing. Five questions need to be known(those questions have to be provided preferable via a WEB portal due to the distributed nature of the development environment):

1. What's new in the build today, what (textual description) has changed compared to the baseline before
2. Are there known (from developers) dependencies in the build. Are possible work arounds suggested?
3. What about interface changes, if yes which ones

4. Which bug-ids, etc. have been tackled by this build/baseline.
 5. What has changed (semantic description) between baselines more older baselines?
- Problems:
 - Interfaces do not match. Interface problems (missing interfaces, interfaces having the wrong semantics)
 - Incompatibilities e.g. due to dependencies on certain versions of libraries, etc. (rare)
 - Problems to fulfill dynamic constraints (like correct sequence of calculation, Trigger times / recurrence not supported by host system, etc.)
 - Modules are not fitting together, because information is missing
 - Reasons:
 - The original specifications for the various components were not checked against each other for consistency.
 - Uncared limitations in the environment.
 - Insufficient variant management
 - Complexity of the system
 - Time pressure for integrating developers
 - Missing and non formalized information
 - Incompatibilities are much harder to solve here than with plain C.
 - When working with models it has to be made sure that only right amount of people have access to the model

Artifacts before implementation

- Problems:
 - A lot of misunderstanding of which information should be given in which artifact on which abstraction level?
 - Requirements from various sources are not consistently channeled into one database
 - Technical requirements are not correctly derived from overall requirements

- A change in overall requirements is not always communicated → derived requirements are not updated
- Keeping the implementation close to the specification.
- Underestimating of the importance of Non Functional Requirements.
- The software architects needs to be involved into functional hardware

Specification vs. Implementation

- Problems:
 - fuzzy specifications can lead to different interpretations from different developers
 - some developer are shy to ask for clarification
 - The programmer requires some free creativity

Semantic inconsistencies between components

- Problems:
 - The type of data exchanged between components is not always the same (e.g. kph / mph).
 - a changed resolution of an integer representation. (i.e. the C-data type stays the same, but the physical interpretation changes).
- Possible Solutions:
 - Naming convention is needed to solve those problems
 - A clear mapping of specification to software implementation is needed

Application-based inconsistencies between components

- Problems:
 - Parallel state machines that do not run synchronously.
 - One module uses obsolete values produced by another module.
- Possible Solutions:
 - A meaningful specification needs to be established, in order to detect such inconsistencies on specification level, but not on implementation level.
 - A meaningful process (describing which role does what and when) needs to be established
 - A meaningful quality control shall be established, also checking the content of specification

Pragmatic inconsistencies between components

- Problems:
 - Run-time restrictions are not always observed overall run time of all modules together exceeds sometimes the maximum allowed value.
 - Functionalities not respecting the defined system timing may fail to be integrated (e.g. because of exceeding the maximum allowed runtime (cooperative blocking time))
 - certain trigger points are not supported in the system (e.g. engine synchronous triggers)
 - Software designed for preemptive/cooperative environment has to be used in a "foreign" context, certain sequencing constraints lead to irresolvable loops of constraints, overall system performance is not compatible with the functionalities to be executed...

Common questions

- Guide Lines are hold? rarely: (0) casually: (2) always: (2)
- Comments:
 - The different development locations are not working with exactly the same tools and not exactly the same target hardware.
 - Per default I would like to prohibit any kind of direct check-in by a developer. Instead I would prefer using the Open Source way where every developer has to achieve the right to become a “committer”. This means every check-in needs to be proven by a responsible guy and only then it will become part of the repository (not yet the next baseline!).
 - For a diagram always a textual description is essential
 - Lack of openness for applying new things

Appendix D.

Summary: Analysis' Results

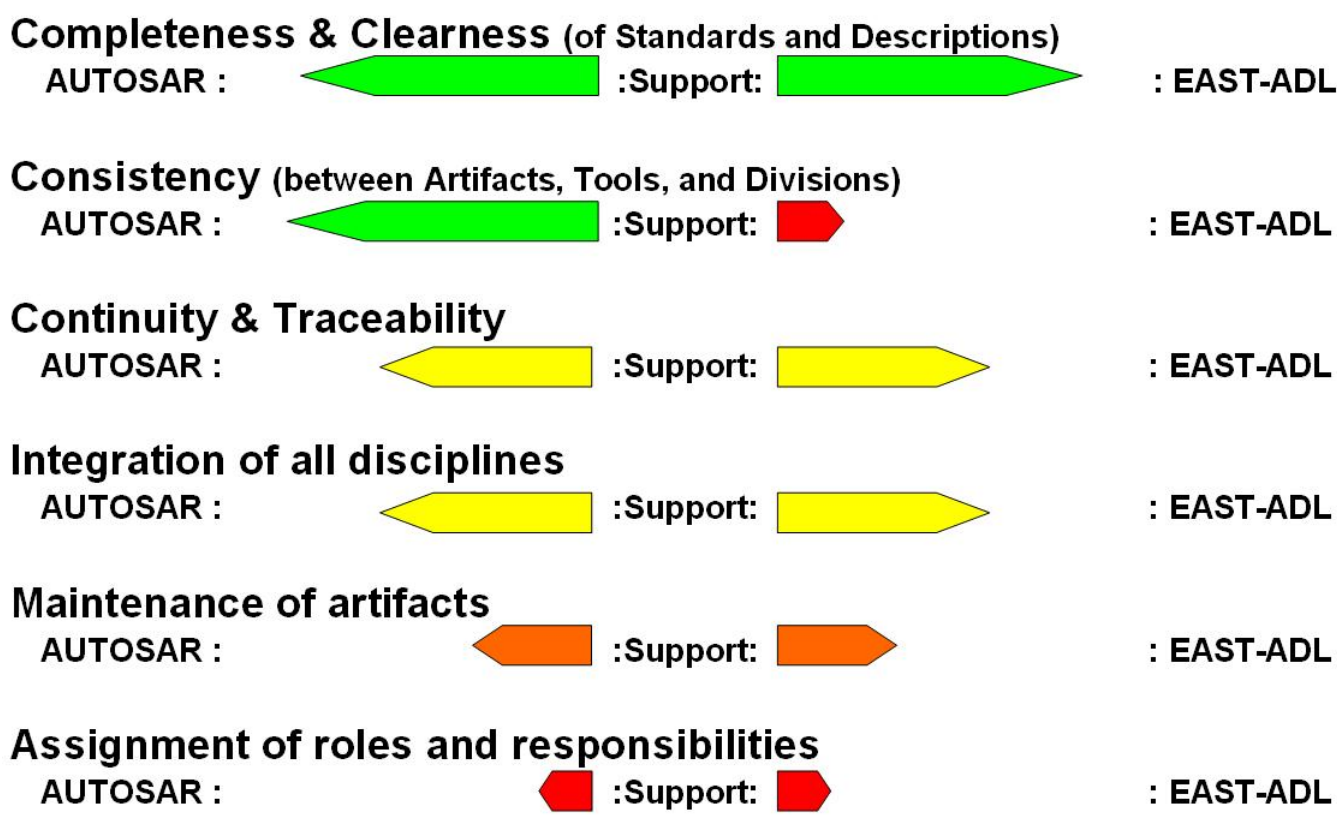


Figure D.1.: Enhanced Basic Conditions by AUTOSAR and EAST-ADL2

	AUTOSAR (static)	AUTOSAR (dynamic)	EAST-ADL2
Syntactic Inconsistencies			
inconsistent data exchange format		SWDataDefProps	
inconsistent interfaces	PortInterface		
inconsistent ports	PortProtoType		
inconsistent signatures	OperationProtoType		
inconsistent data names	DataType		
Semantical Inconsistencies			
numerical inconsistencies		SWDataDefProps	
language inconsistencies	AUTOSAR meta model		EAST-ADL2 meta model
reference system inconsistencies		SWDataDefProps	
Application-based Inconsistencies			
violation of states		partly by ModeDeclarationGroupPrototype	partly by OCL triggerConditions
violation of relations between states and parameter		partly by ModeDeclarationGroupPrototype	partly by OCL triggerConditions
restricted data ranges		Range	ADLDesignDataType
Pragmatic Inconsistencies			
concurrency		ExclusiveArea, Inter-RunnableVariable, ModeDeclarationGroup	partly by OCL triggerConditions
access restriction on extern resources	ComplexDeviceDriverComponentType, EcuAbstractionComponentType, SensorActuatorSoftwareComponentType	RunnableEntity	partly by OCL triggerConditions
timing requirements on hardware		ServiceNeeds, RessourceConsumption	ADLRequirement
absolute timing requirements		TIMMO Project	TimingRequirements
relative timing requirements		TIMMO Project	TimingRequirements
communication pattern	PortInterface		ADLFlowPort, ADLClientServer-Port

Table D.1.: Inconsistencies avoided by EAST-ADL2s' and AUTOSAR static and dynamic semantics

Appendix E.

Case Study: Memory Stack

E.1. Statement Collection: textual representation

A «MemoryStack» is composed of a «Memoryservice», a «Memory Hardware Abstraction», and «Memory Driver».

The «MemoryStack» executes a «Write» Function, which «writes» «Data» «to» «logical block number».

A «Write» Function executes of «NvM_WriteBlock» and «MemIf_Write», which executes either «EE_Write» or «Fee_Write».

The «NVRAM» does «MemIf_Write» by the means of a «NvM_SrcPtr», a «DeviceIndex», and a «BlockId».

The «MemIf» does «MemIf_Write» by the means of a «DataBufferPtr», a «BlockNumbner», and a «DeviceIndex».

A «Pointer», which can be a «NvM_SrcPtr» or a «DataBufferPtr», «points to» «Data», which can be «RAMData».

A «MemoryDriver», which has a «DeviceIndex» and «Name», can be a «FlashDriver» or a «EEP-ROM Driver».

A «logical block number», which is a correlation of «BlockIdentifier» and a «data index», can be a «BlockNumber» or a «BlockId».

The Process «Write» deals with «RAMData» and executes «NvM_Write» then «MemIf_Write» then either «EE_Write» or «Fee_WWrite».

E.1.1. Information Objects

«Data» can be «RAMData». [E.2](#)

A «logical block number» is a correlation of «BlockIdentifier» and a «data index».

A «logical block number» can be a «BlockNumber» or a «BlockId». [E.2](#)

A «MemoryDriver» has a «DeviceIndex» and «Name». A «MemoryDriver» can be a «FlashDriver» or a «EEPROM Driver». [E.2](#)

A «MemoryStack» is composed of a «Memoryservice», a «Memory Hardware Abstraction», and «Memory Driver». [E.2](#)

A «Pointer» can be a «NvM_SrcPtr» or a «DataBufferPtr». [E.2](#)

E.1.2. Functions

The «NVRAM» does «MemIf_Write» by the means of a «NvM_SrcPtr», a «DeviceIndex», and a «BlockId». [E.2](#)

The «MemIf» does «MemIf_Write» by the means of a «DataBufferPtr», a «BlockNumner», and a «DeviceIndex». [E.2](#)

«MemIf_Write» executes either «EE_Write» or «Fee_Write». [E.2](#)

The «MemoryStack» executes a «Write» Function. [E.2](#)

A «Pointer» «points to» «Data». [E.2](#)

«read» «Data» «from» «Pointer». [E.2](#)

«write» «Data» «to» «logical block number». [E.2](#)

A «Write» Function executes of «NvM_WriteBlock» and «MemIf_Write». [E.2](#)

A «Write» Function «reads from» and «writes to». [E.2](#)

E.1.3. Process

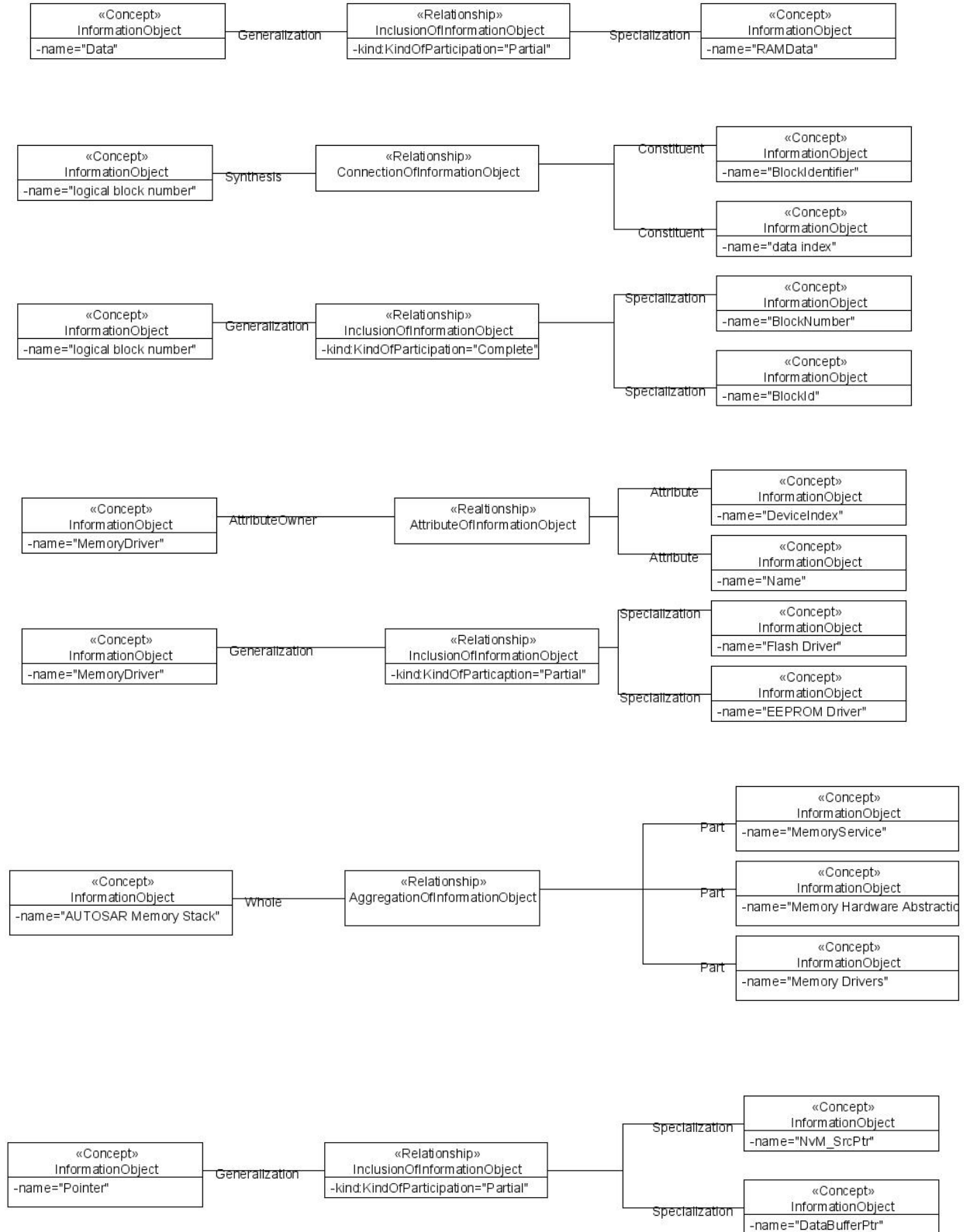
The Process «Write» deals with «RAMData». [E.2](#)

After «NvM_Write» «MemIf_Write» is executed.

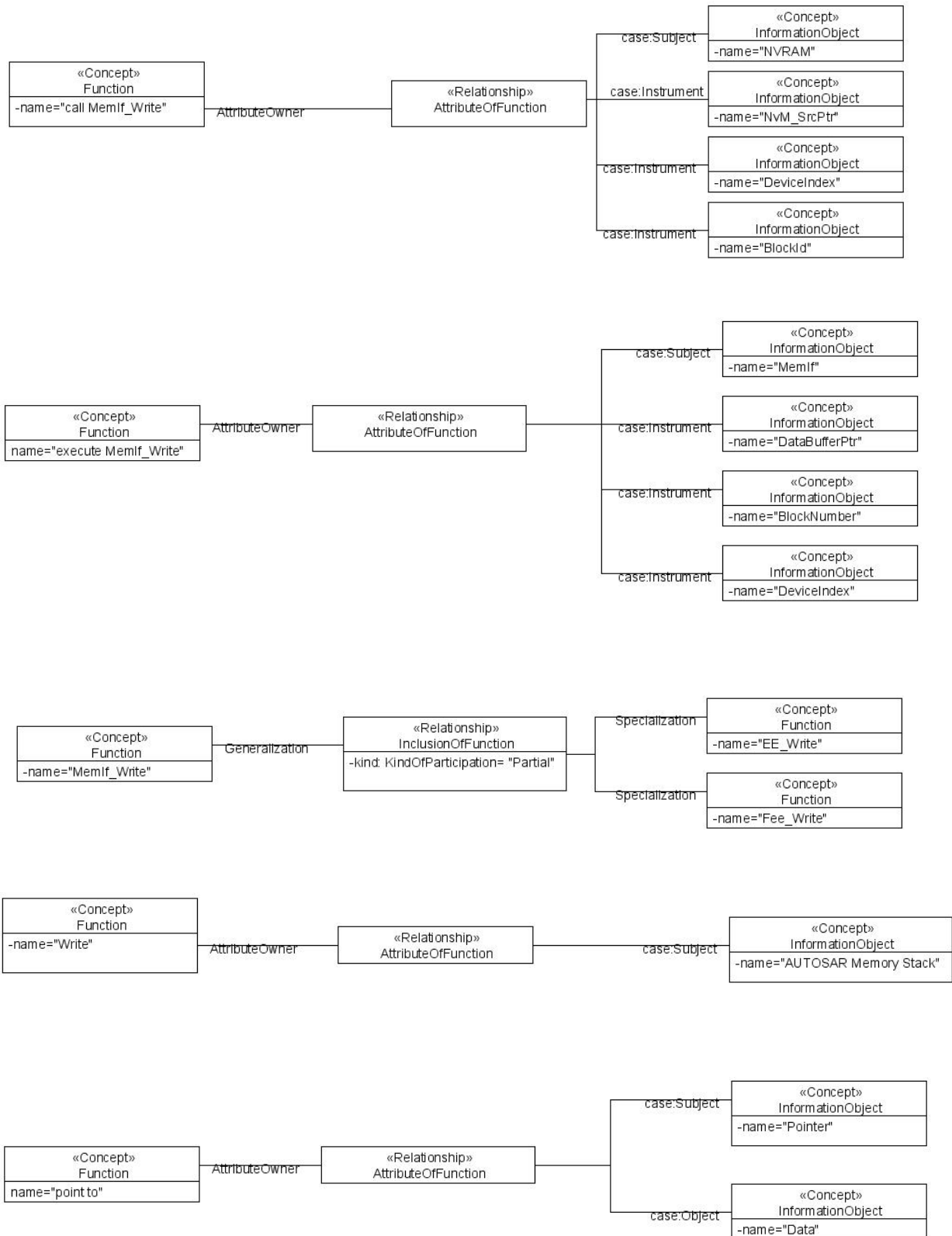
After «MemIf_Write» follows either «EE_Write» or «Fee_Write». [E.2](#)

E.2. Statement Collection: graphical representation

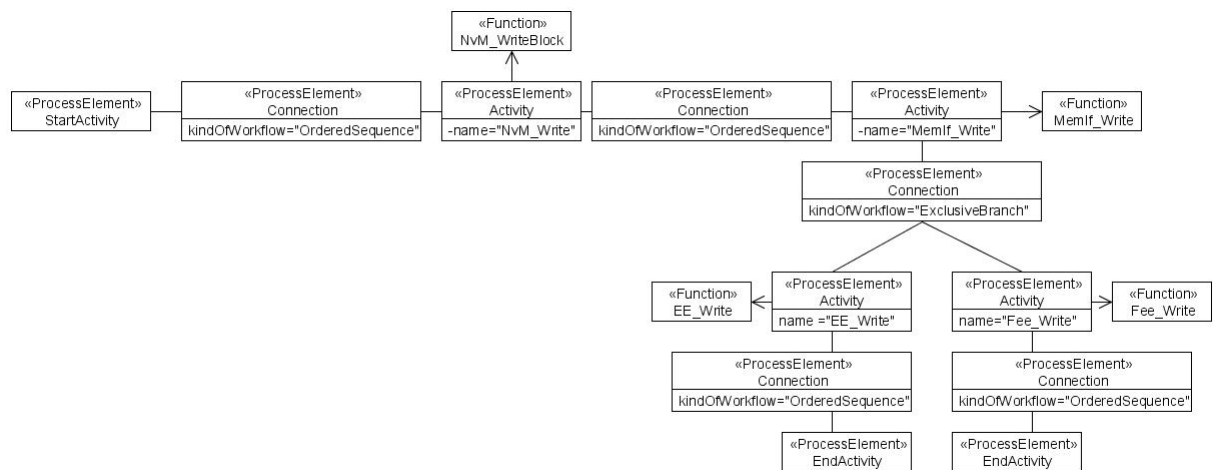
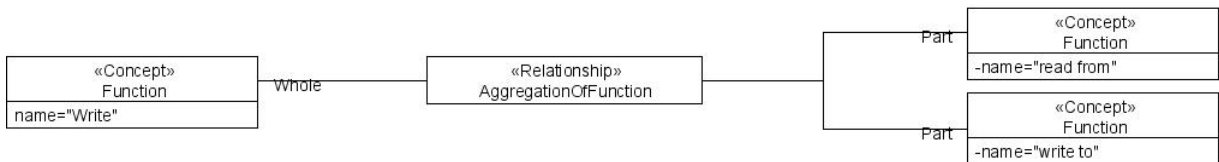
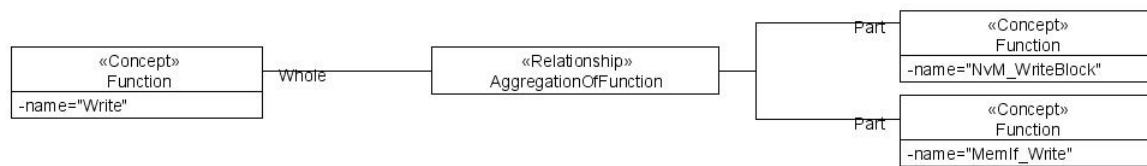
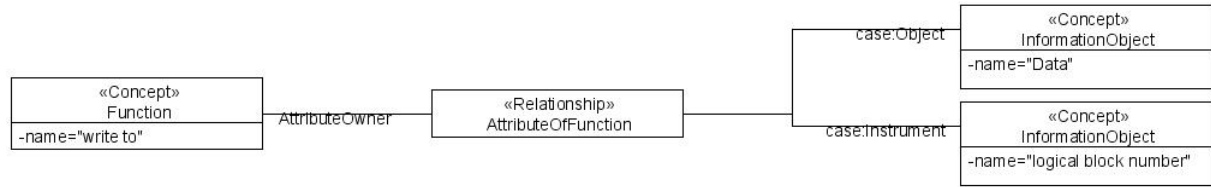
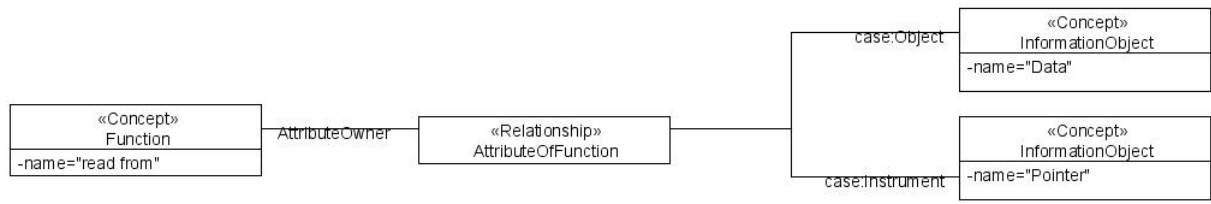
Appendix E. Case Study: Memory Stack

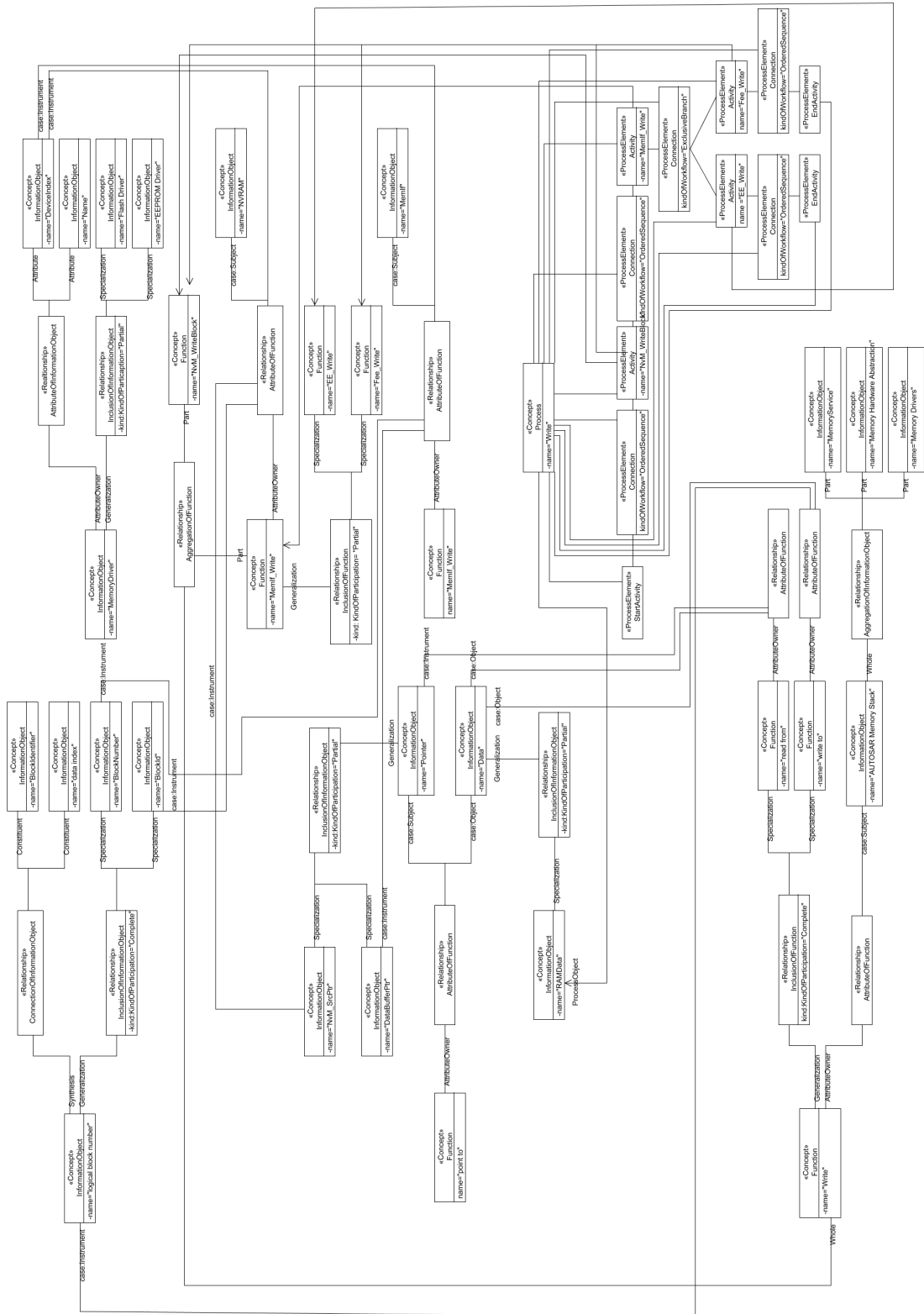


Appendix E. Case Study: Memory Stack



Appendix E. Case Study: Memory Stack





Appendix F.

Bibliography

- [1] Eclipse modeling framework. <http://www.eclipse.org/modeling/emf/>.
- [2] Openarchitectureware. <http://www.openarchitectureware.org/>.
- [3] Society of automotive engineers. <http://www.aadl.info/>.
- [4] Sysml open source specification project. <http://www.sysml.org/>.
- [5] University of stuttgart - institut für automatisierungs- und softwaretechnik. <http://www.ias.uni-stuttgart.de/forschung/kfz.en.html>.
- [6] R. Allen and D. Garlan. A formal basis for architectural connection. In *ACM Trans. Software Eng. and Methodology*, vol. 6, no. 3, pages 213–249, July 1997.
- [7] ATE SST. <http://www.atesst.org/>.
- [8] ATE SST. East adl 2.0 specification. <http://www.atesst.org/>.
- [9] ATE SST. Overview of the east-adl 2. <http://www.atesst.org/>.
- [10] AUTOSAR. <http://www.autosar.org/>.
- [11] AUTOSAR. Autosar ecu configuration. <http://www.autosar.org/>.
- [12] AUTOSAR. Autosar metamodel. <http://www.autosar.org/>.
- [13] AUTOSAR. Autosar methodology. <http://www.autosar.org/>.
- [14] AUTOSAR. Autosar model persistence rules for xml. <http://www.autosar.org/>.
- [15] AUTOSAR. Autosar software component template. <http://www.autosar.org/>.
- [16] AUTOSAR. Autosar technical overview. <http://www.autosar.org/>.

- [17] AUTOSAR. File structure of integrator 2. unpublished.
- [18] AUTOSAR. Layered software architecture. <http://www.autosar.org/>.
- [19] AUTOSAR. Specification for the ecu resource template. <http://www.autosar.org/>.
- [20] AUTOSAR. Specification of bsw module description template. <http://www.autosar.org/>.
- [21] AUTOSAR. Specification of interaction with behavioral models. <http://www.autosar.org/>.
- [22] AUTOSAR. Specification of interoperability of authoring tools. <http://www.autosar.org/>.
- [23] AUTOSAR. Specification of module memory abstraction interface. <http://www.autosar.org/>.
- [24] AUTOSAR. Specification of nvram manager. <http://www.autosar.org/>.
- [25] AUTOSAR. Specification of the system template. <http://www.autosar.org/>.
- [26] AUTOSAR. Sw_c and system modeling guide. <http://www.autosar.org/>.
- [27] AUTOSAR. Validator2: Lesson learned. unpublished.
- [28] S. Becker, A. Brogi, I. Gorton, S. Overhage, and M. Romanovsky, A.; Tivoli. Towards an engineering approach to component adaptation. architecting systems with trustworthy components. Springer Lecture Notes in Computer Science (LNCS) 3938. Dagstuhl Castle, Germany 2006, S.193-215, December 2004.
- [29] J. Bézivin and F. Jouault. Using atl for checking models. In *Electronic Notes in Theoretical Computer Science*, volume 152, pages 69–81, March 2006.
- [30] C. DeJiu, M. Torngren, J. Shi, S. Gerard, H. Lonn, D. Servat, M. Stromberg, and K.-E. Arzen. Model integration in the development of embedded control systems. In *Computer-Aided Control Systems Design, 2006 IEEE International Symposium on*, 2006.
- [31] A. Egyed. Instant consistency checking for the uml. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 381–390, New York, NY, USA, 2006. ACM.
- [32] M. Gagliardi and C. Spera. Some new results in model integration. In *28th Annual Hawaii International Conference on System Sciences*, 1995.
- [33] D. Garlan. An introduction to the aesop system. Internet, July 1995. <http://www.cs.cmu.edu/afs/cs/project/able/www/aesop/html/aesop-overview.ps>.

- [34] A. M. Geoffrion. Structured modeling: Survey and future research directions. <http://www.informs.org/Pubs/ITORMS>, June 1999. .
- [35] M. Gorlick and A. Quilici. Visual programming in the large versus visual programming in the small. In *IEEE Symp. Visual Languages*, pages 137–144, Oct. 1994.
- [36] I. A. W. Group. Recommended practice for architectural description. IEEE P1471/D5.2 Information Technology Draft, December 1999.
- [37] M. Hobelsberger. Vergleich der architekturbeschreibungssprachen east adl und sae adl sowie ihrer beziehung zu autosar. Master's thesis, Fachhochschule Regensburg, 2007.
- [38] F. Jung, Martin; Saglietti. Supporting component and architectural re-usage by detection and tolerance of integration faults.
- [39] M. Jung. *Modelbasierte Gnerierung von beherrschungsmechanismen für Inkonsistenzen in komponentenbasierten Systemen*. Dissertation, technical faculty of the unversity of erlangen-nuernberg, 2006.
- [40] R. Keshav and R. Gamble. Towards a taxonomy of architecture integration strategies. *Foundations of Software Engineering*, 1:98–92, 1998.
- [41] D. Luckham, J. Kenney, L. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using rapide. In *IEEE Trans. Software Eng.*, vol. 21, no. 4, pages 336–355, Apr. 1995.
- [42] J. Magee and J. Kramer. Dynamic structure in software architectures. In *ACM SIGSOFT '96: Fourth Symp. Foundations of Software Eng. (FSE4)*, pages 3–14, Oct. 1996.
- [43] L. Mariani. A fault taxonomy for component-based software. *Electronic Notes in Theoretical Computer Science*, 82:55–56, 2003.
- [44] N. Medvidovic, P. Oreizy, J. Robbins, and R. Taylor. Using object-oriented typing to support architectural design in the c2 style. In *ACM SIGSOFT '96: Fourth Symp. Foundations Software of Eng. (FSE4)*, pages 24–32, Oct. 1996.
- [45] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.*, 26(1):70–93, 2000.
- [46] M. Moriconi and R. Riemenschneider. Introduction to sadl 1.0: A language for specifying software architecture hierarchies. Technical report, SRI-CSL-97-01, SRI Int'l, Mar. 1997.
- [47] OASIS-UDDI-Specification-Technical-Committee. Uddi oasis standard. <http://uddi.xml.org/>.

- [48] OMG. Mof 2.0/ xmi mapping specification, v2.1.1. <http://www.omg.org/technology/documents/formal/xmi.htm>.
- [49] OMG. Query/ view/ transformation, v1.0. <http://www.omg.org/spec/QVT/1.0/>.
- [50] S. Overhage. *Vereinheitlichte Spezifikation von Komponenten: Grundlagen, UnSCom Spezifikation und Anwendungen*. PhD thesis, Universität Augsburg, Wirtschaftswissenschaftliche Fakultät, 2006.
- [51] S. Overhage and P. Thomas. Unscm: A standardized framework for the specification of software components. *Object-Oriented and Internet-Based Technologies: 5th Annual International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World, Net. ObjectDays 2004, Erfurt, Germany*, pages 169–184, 2004.
- [52] R. Racu, A. Hamann, R. Ernst, and K. Richter. Automotive software integration. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 545–550, New York, NY, USA, 2007. ACM.
- [53] C. Reichmann, D. Gebauer, and K. D. Müller-Glaser. Model level coupling of heterogeneous embedded systems. RCBS/MODES04, 2004.
- [54] W. Reif. Lecture: Software engineering. <http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/teaching/>.
- [55] W. Reisig. *Petri nets: an introduction*. Springer-Verlag New York, Inc., New York, NY, USA, 1985.
- [56] M. Shaw, R. DeLine, and G. Zelesnik. Abstractions and implementations for architectural connections. In *Third Int'l Conf. Configurable Distributed Systems*, May 1996.
- [57] A. Terry, R. London, G. Papanagopoulos, and M. Devito. The ardec/teknowledge architecture description language (artek), version 4.0. Technical report, Teknowledge Federal Syst., and U.S. Army Armament Research, Development, and Eng. Center, July 1995.
- [58] W. Tracz. Lileanna: A parameterized programming language . In *Second Int'l Workshop Software Reuse*, pages 66–78, Mar. 1993.
- [59] K. van Heen, N. Sidorova, L. Somers, and M. Voorhoeve. Consistency in model integration. In *Business Process Management*., 2004.
- [60] S. Vestal. Metah programmer's manual, version 1.09. Technical report, Honeywell Technology Center, Apr. 1996.