X-by-Construction Design Framework for Engineering Autonomous and Distributed Real-time Embedded Software Systems

Research and Innovation Actions

Horizon 2020, Topic ICT-50-2020: Software Technologies

Grant agreement ID: 957210

– Deliverable –

D2.1: Modelling Requirements for System's and Interface Specifications

---

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957210.

Document information

Document title:	Modelling Requirements for System's and Interface Specifications
Work package:	WP2
Editor:	VECTOR
Author(s):	VECTOR, KIT, UOP, QUB
Reviewer(s):	UOP, FENTISS
Document type:	Report
Version:	1.0
Status:	Released
Dissemination level:	Public

XANDAR consorti m

No.	Short name	Name	Country
1	KIT	Karlsruher Institut für Technologie	Germany
2	UOP	University of Peloponnese	Greece
3	DLR	Deutsches Zentrum für Luft- und Raumfahrt	Germany
4	AVN	AVN Innovative Technology Solutions Limited	Cyprus
5	VECTOR	Vector Informatik GmbH	Germany
6	BMW	Bayerische Motoren Werke Aktiengesellschaft	Germany
7	QUB	The Queen's University of Belfast	United Kingdom
8	FENTISS	Fent Innovative Software Solutions SL	Spain

Copyright & disclaimer

Document revision history

Version	Date	Comments
1.0	2021-06-29	Publication of the final version.

About this document

This deliverable describes the results of Task T2.1 Requirements for Models, Interfaces & Tools, which derives and specifies the requirements that are necessary to accurately model the non-functional constraints, the safety/security concepts and the system function, network, hardware, and software architecture.

Based on the results of Task T1.1 – Pilot Trials Specifications & Assessment Protocol, which targets analysis and specification of functional and non-functional requirements for the industrial validators, it, at first, introduces the XANDAR development process in Chapter 2. This process shows the general approach of XANDAR and defines the interaction of data, models, and tools. In Chapter 3, the document specifies the essential requirements to accurately model a system's architecture, constraints, and the interfaces to other tools. These requirements represent the result of the further chapters and, thus, the main contribution of the task.

Starting with Chapter 4, the status quo is described. This starts off with an introduction of existing modelling tools that are an essential part of the XANDAR process. Chapter 5 continues with a presentation of existing data models e.g., EAST-ADL, AMALTHEA, to highlight their possible usage to fulfil the previous modelling requirements. Finally, in Chapter 6, the data flow of the XANDAR process is described by presenting the existing interfaces of the aforementioned modelling tools.

With the description of the status quo and a list of requirements, which are directly derived from the work on the demonstrator, a clear view of the existing deficits and, consequently, of the next steps to achieve a X-by-Construction Design framework for Engineering Autonomous & Distributed Real-time Embedded Software Systems is given.

Contents

List of figures

List of tables

Glossary of terms

SiL	Software in the Loop
MiL	Model in the Loop
XML	Extensible Markup Language
UML	Unified Modeling Language
SysML	Systems Modeling Language
M2M	Model-to-Model transformation

1 Introduction

Based on the results outlined in the deliverable D1.1, this deliverable specifies the required extensions necessary to accurately model the non-functional constraints, the safety/security concepts, and the system function, network, hardware, and software architecture. Furthermore, existing data models are evaluated with regards to their potential usage to fulfil the defined modelling requirements. Finally, requirements for extensions to fulfil the aforementioned needs are proposed and specified. Additionally, the requirements for the complementary verification and simulation tools as well as the subsequent code generation processes are specified.

In the Section 2, the XANDAR development process, a means to achieve X-by-construction, is introduced. It starts with the specification of functional and non-functional requirements under consideration of the execution platform and safety & security building blocks. The logical and software architecture, hardware topology, code and system environmental model are the major user-defined artefacts before the integration phase. The results of the process are a functional behaviour for which the functional and non-functional requirements have been verified, or reasons for failure to fulfil the non-functional requirements. Furthermore, the code and the software configuration that fulfil said requirements, are a product of the process.

Section 3 lists requirements towards modelling tools and languages from perspective of the XANDAR toolchain and, in part, motivated by the use cases in the XANDAR project. These shall guide the following discussion of tools, meta-models, and interfaces in terms of their suitability to achieve the goals of the project. The project goals are referenced in the requirements.

The next section introduces the tools and modelling concepts used in the XANDAR toolchain. The Vector product PREEvision is a powerful system modelling tool for various aspects and stages of the automotive development process, complete with its own meta-model albeit largely compatible with AUTOSAR, for example. Another Vector product, the TA Tool Suite, is a specialized set of tools for timing analysis, design, simulation, and verification. An integration of both tools in terms of interfaces is one of the goals in the project. Ptolemy II is an opensource project for functional modelling and simulation. Together, in the XANDAR toolchain, these tools shall enable the users to follow the X-by-construction paradigm.

In Section 5, following the tool introduction, several modelling languages and concepts are introduced to contextualize the XANDAR process as well the modelling requirements defined before. These shall give an overview over the current capabilities of said modelling languages and help define extensions and alterations to support the XANDAR vision. Interoperability of tools is especially important in a toolchain context. Thus, the models are followed by an introduction on the interfaces supported by the aforementioned tools.

Safety and security concepts are inherently part of the XANDAR modelling approach (without explicitly marking them as being safety and security relevant). This means that all safety and security decisions will have been made before starting the modelling process. For this reason, there are no explicit safety and security modelling requirements or planned extensions.

2 XANDAR development process

The XANDAR development process defines the necessary steps to be taken during system development following the XANDAR methodology. In this chapter, the basic concepts behind the X-by-construction process are described and central aspects of the process steps are explained.

Within the XANDAR project, the process is used to align work on toolchains and models. It is in a preliminary stage; details may be updated during the project.

2.1 Central concepts

The XANDAR process is a development process for software-centric embedded systems. It focuses on networked embedded systems in safety-critical domains such as autonomous driving or avionics.

Following the X-by-construction paradigm, it is driven by a precise and formal requirements specification. Design decisions provided by the developer restrict the design space to set the basis for the automation of significant parts of the process. If the tools in the development process fail to identify a suitable solution, the process aborts. Developers making use of this process are then expected to adjust the supplied input artefacts and re-run the process.

To keep the design space manageable, it is envisaged that the toolchain supporting the proposed development process is pre-configured for a certain hardware/software platform. This platform comprises certain hardware modules (e.g. processing units, on-chip networks, peripherals) as well as certain base software (e.g. AUTOSAR, hypervisors, …). While the number and the configuration of platform modules instantiated by the developer can be specified in the input artefacts, the pre-configured nature of the toolchain constraints the component types that are available for such an instantiation. The pre-configuration is defined by the toolchain provider.

The process comprises model-based design approaches to allow for early verification of the correct system functionality. This verification is achieved by deriving a functional system simulation from the developer input. The same input is then used for the generation of the actual system components (i.e. software modules and base software configuration), which are then used as the starting point for the verification of non-functional requirements. This requires semantic preservation, i.e. the guarantee that the transformations applied after functional verification do not modify the functional behaviour and that timing behaviour stays within the bounds set during functional verification.

As a specific mechanism to achieve the X-by-construction property, the process includes a pattern-based approach for realizing non-functional properties such as safety and security. Pre-defined code and model transformations are used to enhance the reliability and resilience of the system to meet the requirements. The developer chooses the transformation patterns to apply from a library of safety/security building blocks. The library itself is provided by the toolchain provider. There can be both platform-independent transformations (e.g. temporal

redundancy) as well as transformations depending on properties of the hardware and base software to be used.

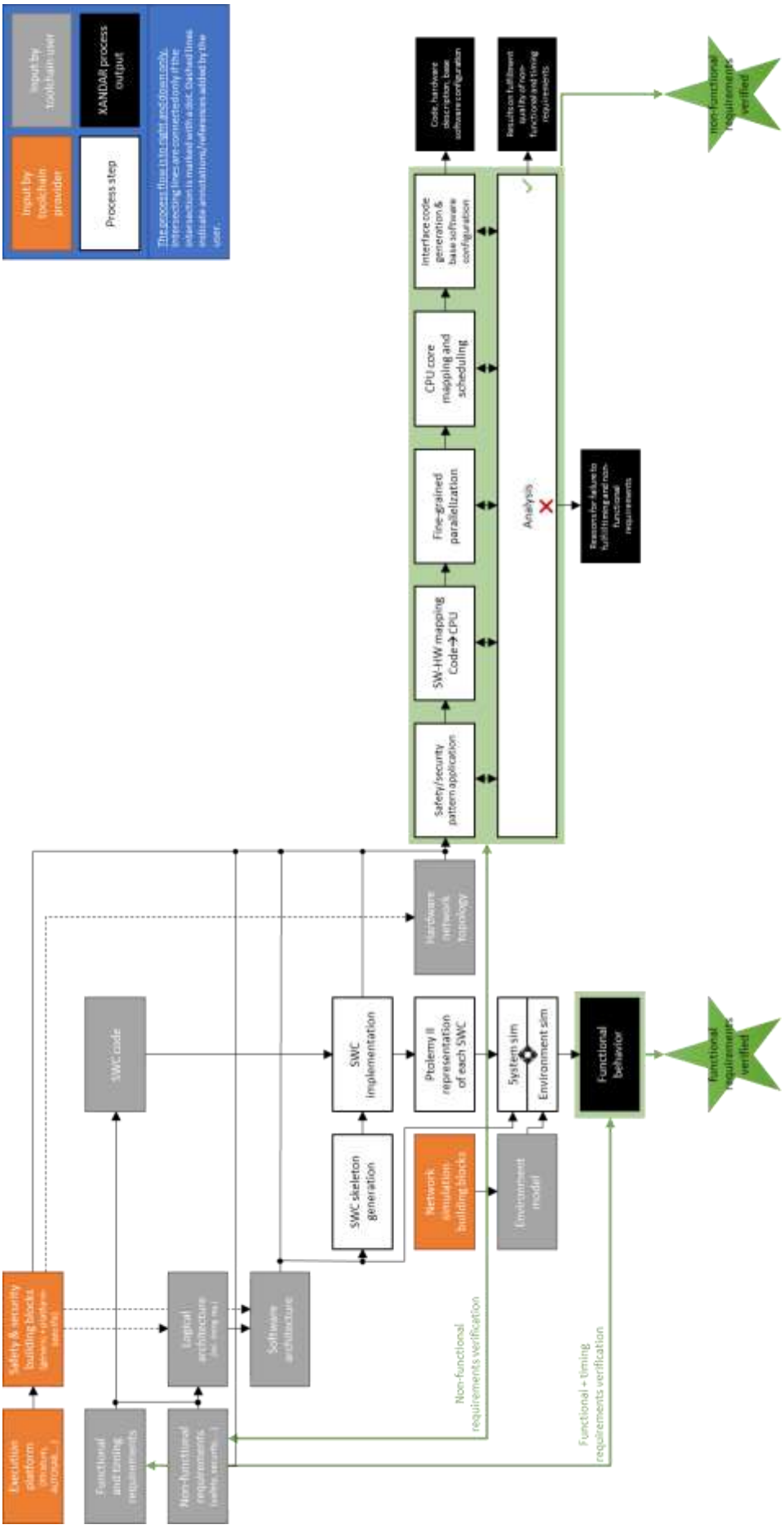

Figure 2-1: XANDAR development process

2.2 Process definition

Figure 2-1 shows the XANDAR development process. Process steps and information artefacts are depicted as blocks while information flow between these blocks is indicated by arrows. Note that information may only flow to the right and down along those arrows. Intersecting lines are not connected unless marked by a dot at the intersection.

The blocks and arrows are grouped by colour. Orange blocks indicate that this information is provided by the toolchain provider. Grey blocks indicate that the information is provided by the toolchain user, i.e. the system developer. White blocks indicate process steps and artefacts which are generated using the toolchain. For these steps, a high degree of automation or toolchain support is envisioned. Black boxes indicate process output artefacts. Finally, green elements indicate verification- and verification-related aspects, such as verification checkpoints.

Concerning the information flow between blocks, black solid lines indicate information flow from one block to another while black dashed lines indicate a mapping of elements from a library to process artefacts. Green solid lines indicate the verification of process artefacts against the respective requirements.

2.2.1 Requirements elicitation

In the beginning of the development process, the requirements towards the system under development are captured. For further usage by the XANDAR process, the requirements are classified as *functional requirements (including timing)* and *non-functional requirements*. The former describes requirements concerning the system functionality, including functionalityrelevant timing requirements (e.g. the requirement of an airbag to fire during a specific time interval after an accident is detected). Non-functional requirements include requirements concerning safety and security, e.g. concerning system reliability, resilience, and isolation requirements, among others.

2.2.2 System architecture

From the requirements, a *logical architecture* is developed. It describes the logical functions envisaged to fulfil the requirements as well as the intended information flows between these functions. The logical architecture comprises a structural description of the system in form of logical functions as well as a behavioural description based on, e.g. state machine diagrams or activity diagrams.

Starting from the logical architecture the developer derives the *software architecture*. It describes all software components in the system as well as their interconnection, i.e. data communication channels between them. Here, a software component (SWC) is a software module that realizes a function in the system under development, e.g. a crash detection module. A more detailed description of the software structure is given in Deliverable 3.1 of the XANDAR project.

In the system architecture, timing requirements can be annotated to logical function blocks or sequences of them. They are annotated in the form of execution and communication budgets

which are considered in the subsequent process steps. They are used for temporal behaviour simulation and need to be fulfilled in the process steps concerning system generation, especially in schedule generation.

With respect to non-functional requirements, safety and security building blocks can be annotated to elements of the logical architecture. They will be applied to the respective elements in a later process step, see Section 2.2.4.1.

2.2.3 Software implementation for model-based behavioural verification

The XANDAR process implements a model-based approach to the verification of functional and timing requirements. The multi-domain simulator Ptolemy II [1] is used to allow for the simulation of the embedded system in its environment. The environment comprises a plant model describing the system under development (e.g. car, aircraft) and – where necessary – an environment model describing its surroundings (e.g. road, wind, other vehicles). Since the environment may involve other systems, connected via wired or wireless networks, a network simulation based on pre-defined building blocks can be used. The resulting simulation output is used to verify that functional and timing behaviour of the system is as specified by the requirements.

While enabling such a simulation-based verification of functional and timing requirements, the XANDAR process offers high flexibility in the implementation of the functionality. Especially, it allows for different types of implementations, such as traditionally-developed and legacy software as well as AI approaches. This flexibility is achieved by integrating developerprovided code and AI models into the simulation before deploying it on the target platform.

To enable the functional simulation of the system, the modelled software architecture is used to generate a *SWC skeleton* for each SWC. Such skeletons are code files with a pre-defined structure, into which the developer can insert the *SWC code*, which results in the *SWC implementation*. When implementing the SWCs, the developer adheres to the programming model defined in [D3.1]. It demands the use of an SWC API, which enables the SWC implementations to be transformed into *Ptolemy II representations of each SWC* and to be deployed on the platform. The Ptolemy II representations can be included in the simulator, where all SWCs are interconnected based on the communication relations defined in the software architecture.

If the simulation indicates that the functional behaviour of the system is correct, the system generation part of the XANDAR process is started. To ensure that the functional and timing requirement verification by simulation holds true, the subsequent process steps need to guarantee semantic preservation, i.e. to preserve the functional behaviour of the code as simulated, and preservation of the timing behaviour. Since the timing behaviour is annotated as timing bounds in the system design, the process steps must guarantee that these timing bounds are met.

2.2.4 System generation

When the result of the SiL simulation is in line with the functional requirements, the system is prepared for deployment on the target hardware and optimized for performance and

safety/security. This is done based on the non-functional requirements, the software architecture, the hardware and network topology, and the service implementations. In the system generation process steps, these input artefacts are forwarded in a pipelined manner while the system design is refined in multiple steps. Thereby, additional requirements and limitations concerning mapping, scheduling, and isolation, for example, are generated and later considered during the respective mapping and scheduling steps.

After each step, *analysis* steps are made to determine if all non-functional requirements and limitations relevant to the process step were met and to filter out solutions which do not meet them. In case that no solution that fulfils the non-functional requirements and limitations can be found based on the provided inputs, the process aborts and emits information to help the developer change inputs and reiterate the process. Due to the refinement approach in system generation, in the final step, the analysis can ensure that all non-functional requirements and limitations are met.

2.2.4.1 Pattern-based approach to non-functional requirements fulfilment

To help fulfil non-functional requirements during system generation, a pattern-based approach is used. A library of *safety and security building blocks* is provided by the toolchain provider. These building blocks include both platform-independent and platform dependent templates. While platform-independent approaches such as temporal redundancy or data encryption can be realized on any execution platform, platform-dependent approaches make use of base software services or hardware characteristics of a specific platform to enhance system reliability or security. Examples include hardware watchdogs, monitoring components, and isolation features, such as Trusted Execution environments.

During system design, the developer maps these templates to elements of the system architecture. These annotations are processed by the *safety/security pattern application* process step, where the respective templates are applied to the set of implemented tasks. This may lead to the refinement of scheduling and mapping requirements to ensure isolation and independence, as well as to the transformation and extension of the software architecture by generation of additional services, e.g. for redundancy. And it may lead to base software configuration requirements, e.g. for monitor configuration. The refined configuration is then passed on to the CPU-level software-hardware mapping process block.

As the process technology continues to shrink, a large number of SRAM bit cells in on-chip caches is expected to be faulty. As a result, many Cache Fault-Tolerance (CFT) techniques have been developed to ensure error free execution in the presence of memory faults. This is a widely studied area where many alternative CFT organizations have been proposed. However, the impact of compiler transformations in the performance of faulty caches as well as performance predictability in the presence of faults has been largely disregarded. It is important to note that performance for a given program may vary noticeably depending on the faulty bit location. The number of faults (hard errors) in a system will be considered as a design pattern in the XANDAR project.

2.2.4.2 SW-HW mapping and parallelization

For the coarse-grained parallelization, the SW-HW mapping can be realized either by the XANDAR mapping system or by both the XANDAR mapping system and the user. In the first case, the XANDAR and the RTE mapping system will map the software routines on the available processors by considering a number of parameters such as the routine's characteristics (e.g., workload, data access patterns), the characteristics of the routines that have already been mapped on this processor, and the hardware architecture details (e.g., type of processors, number of CPU cores). The mapping refers to coarse grain mapping (processor level).

In the second case, the user can ease and/or improve the mapping process by providing scheduling hints/guidelines; the mapping hints can be either explicit (e.g., manually specify the processor to run a specific routine) or implicit (the user provides software functions which are written (or annotated) by using a parallel programming language already). For example, consider the case where the user provides a CUDA-written function; in this case, the user specifies (implicitly) that this function will run on a GPU, without specifying which. In this case, the exploration space is reduced, and the mapping system has fewer solutions to explore.

After the SW-HW mapping process has ended, the routines are further optimized by enabling fine grain parallelization (multithreading and vectorization) and loop transformations. Again, this is realized either just by the XANDAR fine-grain parallelization system or by both the XANDAR fine-grain parallelization system and the user. In the first case, we will consider developing auto-annotation methods or using existing solutions such as the one provided by the LLVM framework. In the second case, the user can provide the appropriate OpenMP/OpenACC annotations to parallelize/vectorize the code.

The XANDAR toolchain also supports loop transformations, such as loop tiling or register blocking, to allow for improved performance or safety, e.g., faulty caches (more details about this can be found in Section 2.2.4.1). To this end, the loop transformations will be applied via annotation-based languages such as OpenMP.

2.2.4.3 Interface code generation and base software configuration

In the final step of the XANDAR process, the system defined in the previous steps is linked to the base software and platform. This includes the configuration of the base software based on the previously determined mapping and scheduling aspects, communication link configuration, as well as further configurations targeting non-functional requirements such as monitor configuration, among others. Wrapper code is generated to adapt the services implemented against the XANDAR Service API to the base software runtime API.

This step outputs code and base software configurations which can then be used to build the target software using the base software toolchain.

3 Requirements regarding modelling and interfaces

The following is a collection of requirements towards modelling tools and their interfaces, including behavioural, functional, non-functional (timing), and code generation aspects.

Safety and security concepts are inherently part of the XANDAR modelling approach (without explicitly marking them as being safety and security relevant). This means that all safety and security decisions will have been made before starting the modelling process. For this reason, there are no explicit safety and security modelling requirements or planned extensions.

The requirements are given in a structured manner. Where possible, a realization proposal has been given by the requirement author for consideration by the project consortium. Since this field is optional it is empty in some requirements. These proposals are non-binding and subject to change. For reference, the target domain and overall project objectives are given since the requirements are aligned with the general goals of XANDAR. Note that some of these requirements are derived from objective and metric considerations in D1.1. Others are more detailed statements about planned modelling and interface extensions. The original requirement's authors are stated here to see whom to contact in case there are questions.

ID	**REQ_01**	**Export of Event Chains**		
Source		**Domain**	**Objective**	**Document**
		Automotive	OBJ3, OBJ5	D2.1
Author		Vector		
Category		Timing		
Description		To validate timing requirements, event chains should be provided in a standard format to process them in validation tools.		
Rationale		For subsequent validation of timing requirements.		
Related Requirements				
Realization Proposal		Use AUTOSAR TimingExtensions as export format.		
Attributes		**Priority**	**WP/Task**	
		medium	WP2/T2.1	

ID	**REQ_02**	**Modelling Support for Event Chains**		
Source		**Domain**	**Objective**	**Document**
		Automotive	OBJ3, OBJ5	D2.1
Author		Vector		
Category		Timing		
Description		To validate timing requirements, event chains should be supported for modelling. Semantically, they should be in line with AUTOSAR.		
Rationale		For subsequent validation of timing requirements.		
Related Requirements				

Realization Proposal		
Attributes	**Priority**	**WP/Task**
	high	WP2/T2.2

ID **REQ_03**	**Modelling Support for Budget Requirements**		
Source	**Domain**	**Objective**	**Document**
	Automotive	OBJ3, OBJ5	D2.1
Author	Vector		
Category	Timing		
Description	It shall be possible to annotate required budgets to logical functions (i.e. in number of instructions, time, or MIPS).		
Rationale	For subsequent validation of timing requirements.		
Related Requirements			
Realization Proposal			
Attributes	**Priority**	**WP/Task**	
	medium	WP2/T2.2	

ID **REQ_04**	**Modelling Support for Communication Size**		
Source	**Domain**	**Objective**	**Document**
	Automotive	OBJ3, OBJ5	D2.1
Author	Vector		
Category	Timing		
Description	It shall be possible to annotate the size to logical signals, e.g., in bits or bytes.		
Rationale	For subsequent validation of timing requirements.		
Related Requirements			
Realization Proposal			
Attributes	**Priority**	**WP/Task**	
	low	WP2/T2.2	

ID	REQ_05	Modelling Support for Sensor Sampling Rate		
Source		Domain	Objective	Document
		Automotive	OBJ3, OBJ5	D2.1
Author		Vector		
Category		Timing		
Description		It shall be possible to annotate the sampling rate at which a sensor output is provided on a sensor function, e.g., a sampling period, or a frequency.		
Rationale		For subsequent validation of timing requirements.		
Related Requirements				
Realization Proposal				
Attributes		Priority	WP/Task	
		medium	WP2/T2.2	

ID	REQ_06	Modelling Support for Function Activation Pattern		
Source		Domain	Objective	Document
		Automotive	OBJ3, OBJ5	D2.1
Author		Vector		
Category		Timing		
Description		It shall be possible to annotate the activation pattern(s) with which a logical function shall be activated, e.g., a period, or frequency.		
Rationale		For subsequent validation of timing requirements.		
Related Requirements				
Realization Proposal				
Attributes		Priority	WP/Task	
		medium	WP2/T2.2	

ID	REQ_07	Modelling Support for Communication Capacity		
Source		Domain	Objective	Document
		Automotive	OBJ3, OBJ5	D2.1
Author		Vector		
Category		Timing		
Description		It shall be possible to annotate the provided communication capacity of communication connections like buses, e.g., in KBitPerSecond.		
Rationale		For subsequent validation of timing requirements.		
Related Requirements		REQ_04, REQ_05, REQ_06		

Realization Proposal		
Attributes	Priority	WP/Task
	low	WP2/T2.2

ID	REQ_08	Modelling Support for Computing Unit Capacity		
Source		Domain	Objective	Document
		Automotive	OBJ3, OBJ5	D2.1
Author		Vector		
Category		Timing		
Description		It shall be possible to annotate the provided computation capacity of computation units like microprocessors or microcontrollers, e.g., in MIPS.		
Rationale		For subsequent validation of timing requirements.		
Related Requirements		REQ_03, REQ_05, REQ_06		
Realization Proposal				
Attributes		Priority	WP/Task	
		medium	WP2/T2.2	

ID	REQ_09	Modelling of the software component network		
Source		Domain	Objective	Document
		Automotive/Avionic	OBJ3	D2.1
Author		KIT		
Category		System design		
Description		To facilitate model-based system synthesis, the software architecture shall capture all relevant software components (SWCs) as well as their input ports, output ports, low-level device ports, and interconnections.		
Rationale		The envisaged XANDAR design methodology (see D3.1) depends on a model representation of the SWC network.		
Related Requirements		REQ_10		
Realization Proposal				
Attributes		Priority	WP/Task	
		high	WP2/T2.2	

ID	REQ_10	Modelling of runnables and activation policies		
Source		Domain	Objective	Document
		Automotive/Avionic	OBJ3	D2.1
Author		KIT		
Category		System design		
Description		In the software architecture, it shall be possible to model a list of runnables and their activation policies for each SWC. Therefore, both time-based and data-based activation policies shall be supported.		
Rationale		The envisaged XANDAR design methodology (see D3.1) depends on a model representation of the internal SWC architecture.		
Related Requirements		REQ_09, REQ_06		
Realization Proposal				
Attributes		Priority	WP/Task	
		medium	WP2/T2.2	

ID	REQ_11	SWC-based safety pattern specification		
Source		Domain	Objective	Document
		Automotive/Avionic	OBJ2, OBJ3, OBJ5	D2.1
Author		KIT		
Category		Safety		
Description		Library patterns related to the safety of functions implemented in software shall be specifiable as part of the relevant SWC entry in the software architecture model.		
Rationale		To apply a pattern-based safety mechanism during the system generation phase, an unambiguous description of its nature is necessary.		
Related Requirements		REQ_09		
Realization Proposal				
Attributes		Priority	WP/Task	
		medium	WP2/T2.3	

ID	REQ_12	Software architecture exchange format		
Source		Domain	Objective	Document
		Automotive/Avionic	OBJ3	D2.1
Author		KIT		
Category		System design		
Description		Software architecture information related to SWCs, applied safety patterns, SWC ports, their interconnections, runnables and corresponding activation policies shall be available in a textual exchange format with precise syntax and semantics.		
Rationale		The referenced model artefacts need to be available in a format that can be processed by the toolchain implementing the XANDAR design methodology (according to D3.1).		
Related Requirements		REQ_09, REQ_10, REQ_11		
Realization Proposal		XML with a particular DTD is a possible candidate.		
Attributes		Priority	WP/Task	
		high	WP2/T2.1	

ID	REQ_13	Ptolemy II representation of the environment model		
Source		Domain	Objective	Document
		Automotive/Avionic	OBJ5, OBJ6, OBJ7	D2.1
Author		KIT		
Category		Behaviour		
Description		The environment model to verify and validate functional requirements against needs to be available as a Ptolemy II model. In this model, usage of directors must be limited to the subset explicitly supported by the XANDAR toolchain.		
Rationale		A suitable Ptolemy II model of the system under consideration will be automatically derived and shall be integrated into the environment model for the purposes of functional verification.		
Related Requirements				
Realization Proposal				
Attributes		Priority	WP/Task	
		high	WP2/T2.5	

ID	REQ_14	Modelling support for UML activities		
Source		Domain	Objective	Document
		Automotive/Avionic	OBJ3	D2.1
Author		Vector		
Category		Behavioural modelling		
Description		Modelling UML activities and elementary actions shall be provided on different electric/electronic abstraction layers (e.g. logical architecture).		
Rationale		Description of various aspects of the system with different levels of granularity.		
Related Requirements				
Realization Proposal		Provide modelling capabilities for UML activities and their actions for various electric/electronic components (e.g. logical functions, ECUs, ...).		
Attributes		Priority	WP/Task	
		high	WP2/T2.2	

ID	REQ_15	Integrated modelling of UML activity diagrams		
Source		Domain	Objective	Document
		Automotive/Avionic	OBJ3, OBJ5	D2.1
Author		Vector		
Category		Behavioural modelling		
Description		Model-based behavioural description for activities based on UML and SysML concepts shall be provided.		
Rationale		Support of behavioural specification using activity diagrams complementary to state machine diagrams.		
Related Requirements		REQ_14		
Realization Proposal		Provide integrated modelling of activity diagrams with an graphical editor.		
Attributes		Priority	WP/Task	
		high	WP2/T2.2	

ID	REQ_16	Integration of UML activity descriptions in state machines		
Source		Domain	Objective	Document
		Automotive/Avionic	OBJ3, OBJ5	D2.1
Author		Vector		
Category		Behavioural modelling		
Description		An integration of UML activity descriptions in state machines shall be provided.		

Rationale	Detailing the state and state transition behaviour with activities.	
Related Requirements	REQ_14, REQ_15	
Realization Proposal	Provide modelling of activities for state behaviour and state transition.	
Attributes	**Priority**	**WP/Task**
	high	WP2/T2.2

ID	**REQ_17**	**Coupling of UML activities with electric/electronic architecture model.**		
Source		**Domain**	**Objective**	**Document**
		Automotive/Avionic	OBJ3, OBJ5	D2.1
Author		Vector		
Category		Behavioural modelling		
Description		A coupling of UML activities with data from the electric/electronic architecture model shall be provided.		
Rationale		Using existing port and data descriptions of electric/electronic components in behavioural models.		
Related Requirements		REQ_14, REQ_16		
Realization Proposal		Provide an interface concept for behaviour and domain-specific model.		
Attributes		**Priority**	**WP/Task**	
		medium	WP2/T2.2	

4 Modelling tools used in XANDAR

4.1 Vector PREEvision

PREEvision is a tool for model-based development of distributed, embedded systems in the automotive industry and related fields [2]. This engineering environment supports the entire technical development process in a single, integrated application.

PREEvision offers comprehensive functions for both classic and service-oriented architecture development, requirements management, communication design, safety-related system design, AUTOSAR system and software design as well as wiring harness development. PREEvision supports the tried-and-tested system engineering principles of abstraction, decomposition, and reuse here.

This integrated and model-based approach enables:

- Early evaluation of E/E architectures
- Consistent requirements and test management
- Function-driven development

- Software and communication design according to AUTOSAR
- Model-based wiring harness development
- Efficiency through concepts such as reuse and product line and variant management
- Consistent design in a single tool
- Parallel work on a common database from multiple locations (engineering backbone)

Modelling layers and editors

PREEvision's data model is based on a domain-specific graphical language and allows the description of all the essential elements of an E/E architecture including requirements, the logical design, the software and hardware implementation and also a vehicle's geometry. The data model is structured by different modelling layers based on different levels of abstraction. For each layer PREEvision provides specific diagrams or table editors. Relations between architecture layers give a system wide and continuous traceability.

Different views provide specific information about single artefacts or about parts of the E/E architecture. The Property View, for example, provides information about the properties of an artefact, such as, attributes and relations, whereas the Set Content View displays the content of container artefacts.

Artefacts can be created and displayed in a hierarchical structured tree view or in diagrams:

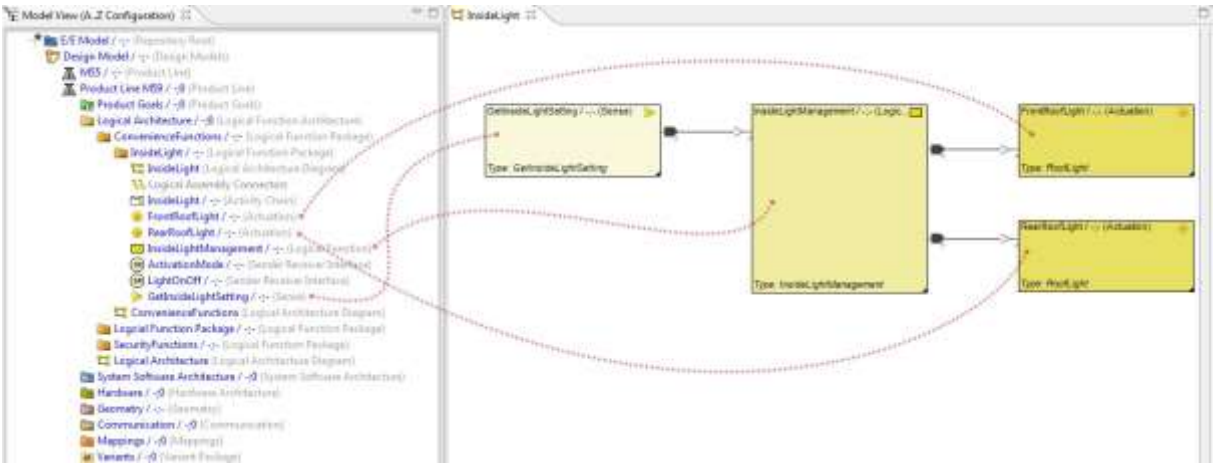

Figure 4-1: PREEvision: Model view and logical architecture diagram

Traceability

Artefacts in different layers can be related to each other via links and mappings. This allows the traceability through all modelling layers. Customer Features describe the customer perceivable characteristics of a vehicle. They may be refined by Requirements that also define the functional and non-functional needs. In the Logical Function Architecture, the logical functionality of the system can be modelled. The logical design is implemented in the System Software Architecture and the Hardware Architecture which can be mapped to the vehicle's topology.

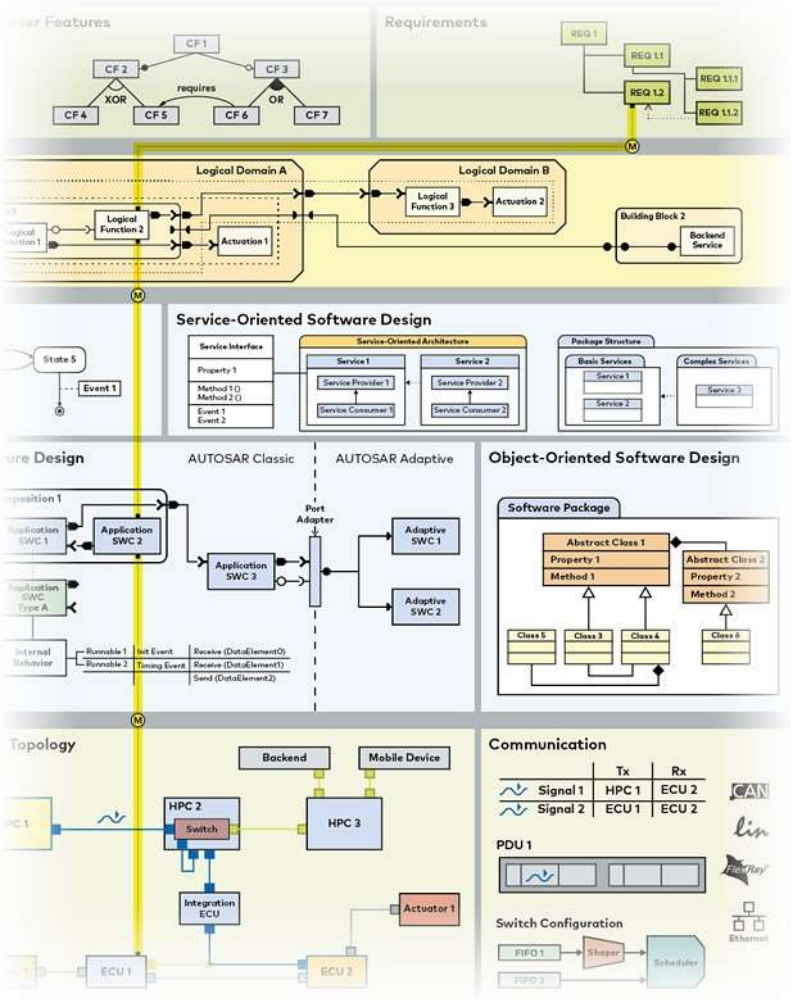

Figure 4-2: PREEvision: Different layers and traceability by mappings

Reuse

The reuse of development artefacts ensures efficient development across product lines. In addition to individual artefacts, entire system sections can be reused.

File management

PREEvision provides an integrated file handling to store additional documents, graphics, or building plans.

Import and export

Standard interfaces such as AUTOSAR, KBL, and ReqIF and scriptable interfaces enable available data to be quickly merged to a common model via import. Via export, the model or dedicated parts of the model can be distributed, for example, to collaborate with partners.

Routing

With the routing mechanisms, an automation concept is available that automatically generates communication for sublayers. After modelling the top design, such as Customer Features and

Logical Function Architecture, routings, such as the signal router or the wiring harness router, can be used to let the system do the next steps automatically.

Product lines and variants

The variant management provides a highly configurable mechanism to model variants based on the E/E architecture. Product lines of an E/E architecture are modelled in PREEvision as 150%-model, that means the model includes all components that can be tailored to different subsets. These subsets – the variants – are representing specific products of one product line.

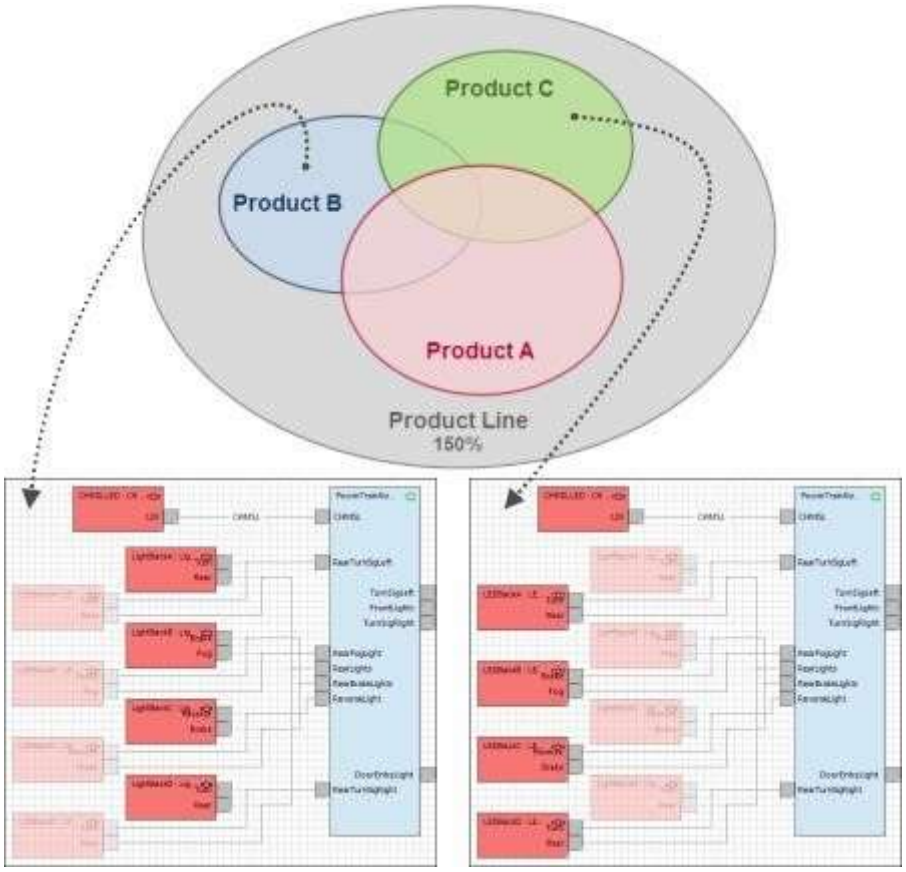

Figure 4-3: PREEvision: Example for product lines and variants

Analysis and evaluation

The success of an architecture can be evaluated based on different optimization goals, such as weight, costs, communication demand, robustness, etc.

PREEvision provides a framework for metric development and the calculation as well as an M2M model transformation technology to create rules, for example, for consistency checks checking the:

- coverage of all requirements by technical implementation
- structural integrity, for example, are all controllers connected?
- quality of the implementation
- consistency and validity of the architecture

System documentation

PREEvision contains a report generator for the system documentation. Besides diagrams or textual artefacts, the report generator processes model queries as well as metric results and produces documents specified by customizable templates. Thus, document generation is not only automated, but it can be designed in a process-specific manner. Finally, complete reports of fully developed architectures can be generated.

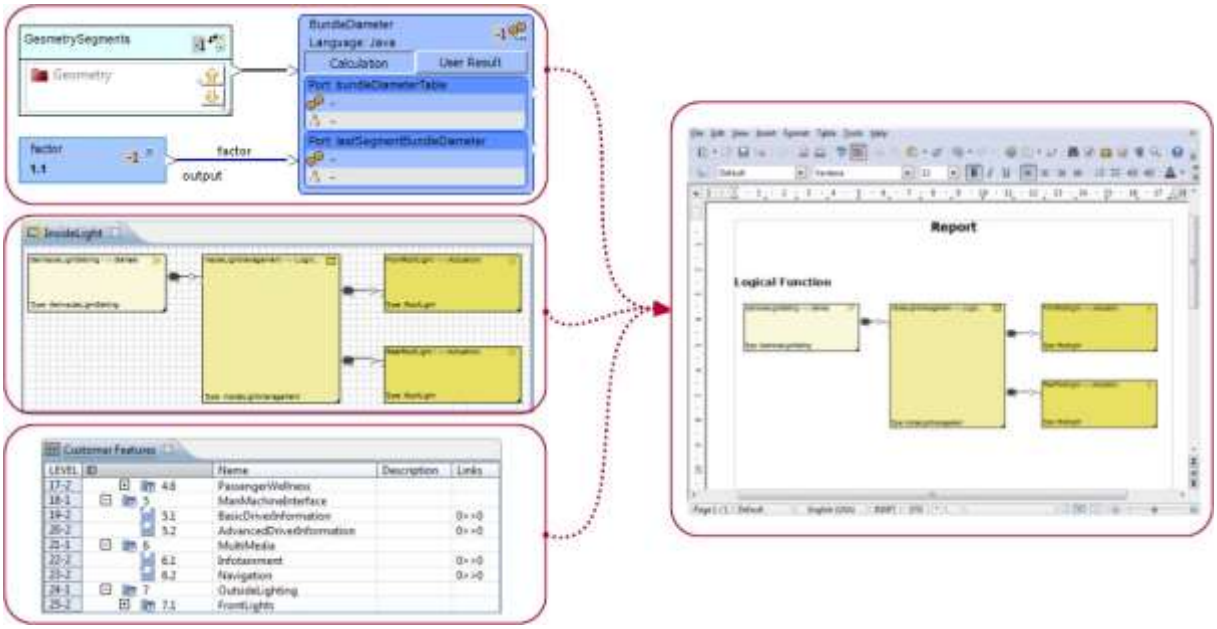

Figure 4-4: PREEvision: Generation of reports for system documentation

Change and release management

In the change and release management, available engineering artefacts as well as tickets for defects and changes can be put under planning control. Thus, the planning status can be automatically synchronized with the engineering progress and provides full traceability of time planning, resource planning and responsibility for engineering artefacts. Instant reports can be created for tracking the development progress and maturity.

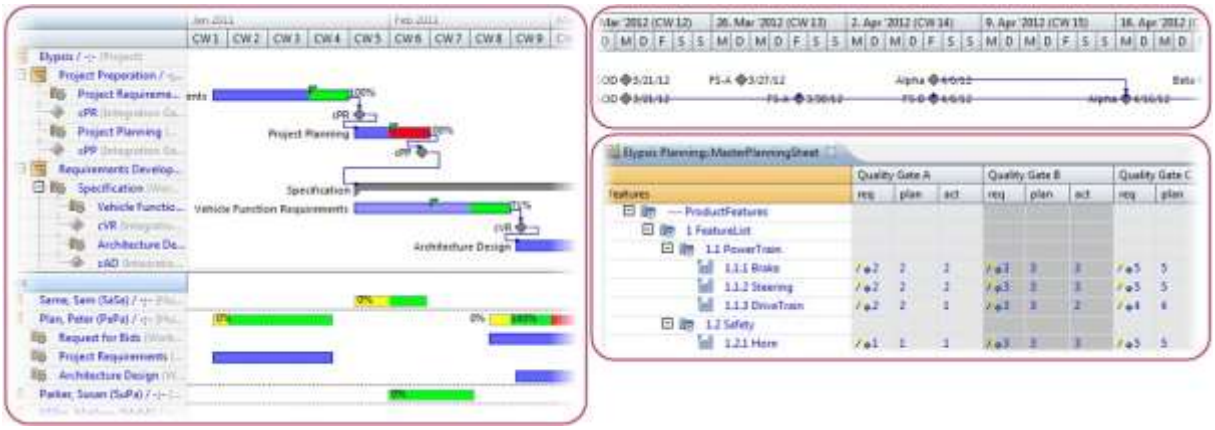

Figure 4-5: PREEvision: Change and release management

Design of safety-related systems

The requirements of ISO26262 for functional safety include responsibilities, development processes, documentation, and technologies for the development of safety-relevant systems. To minimize the effort for development and maintenance of safety-related systems as per ISO 26262, PREEvision offers consistent development support for the entire safety process, from system design to the safety case.

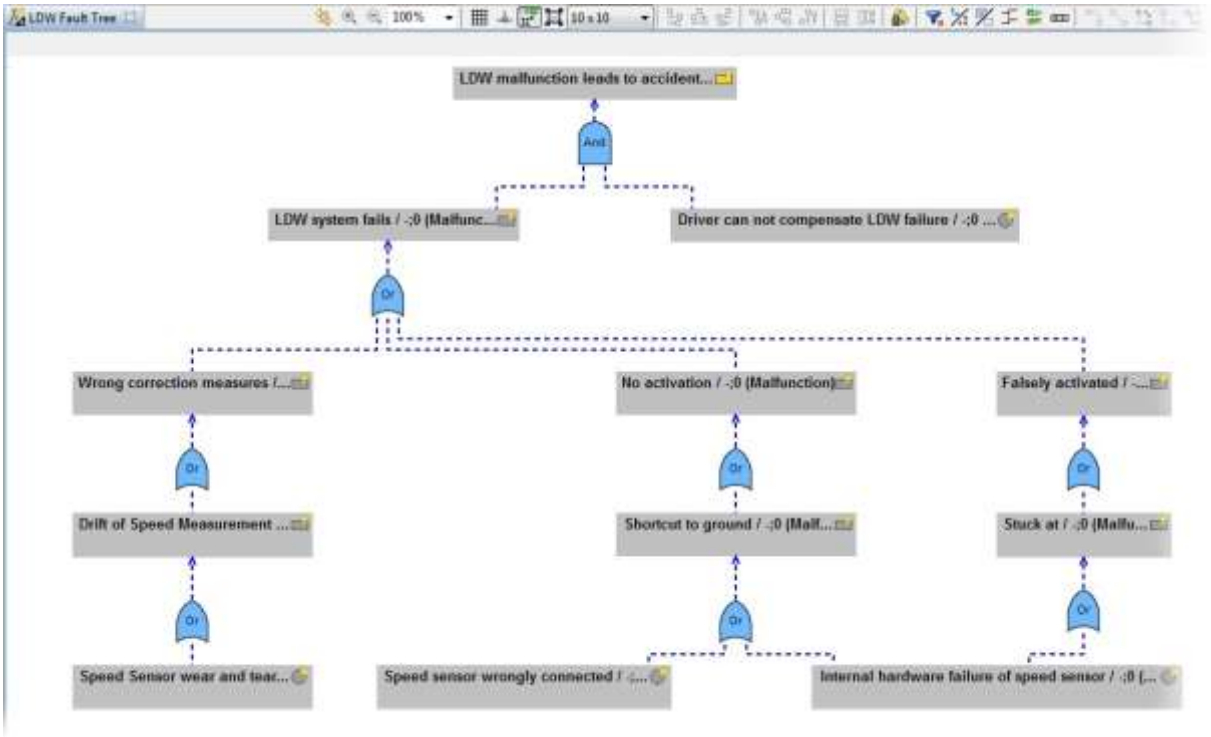

Figure 4-6: PREEvision: Example of a fault tree in a malfunction diagram

Test engineering and test management

PREEvision supports test engineering, control of tests, and the management of accumulated test data across the entire E/E development process. Test scenarios that cover all product requirements can be developed as test specifications and test cases. PREEvision supports the implementation of automatic test scripts and manual test sequences as well as the planning of tests. The results of the automatic test cases are imported from test reports of the execution environment. With manual tests, results are entered directly in PREEvision. The cockpit view provides test managers with the status of all test projects.

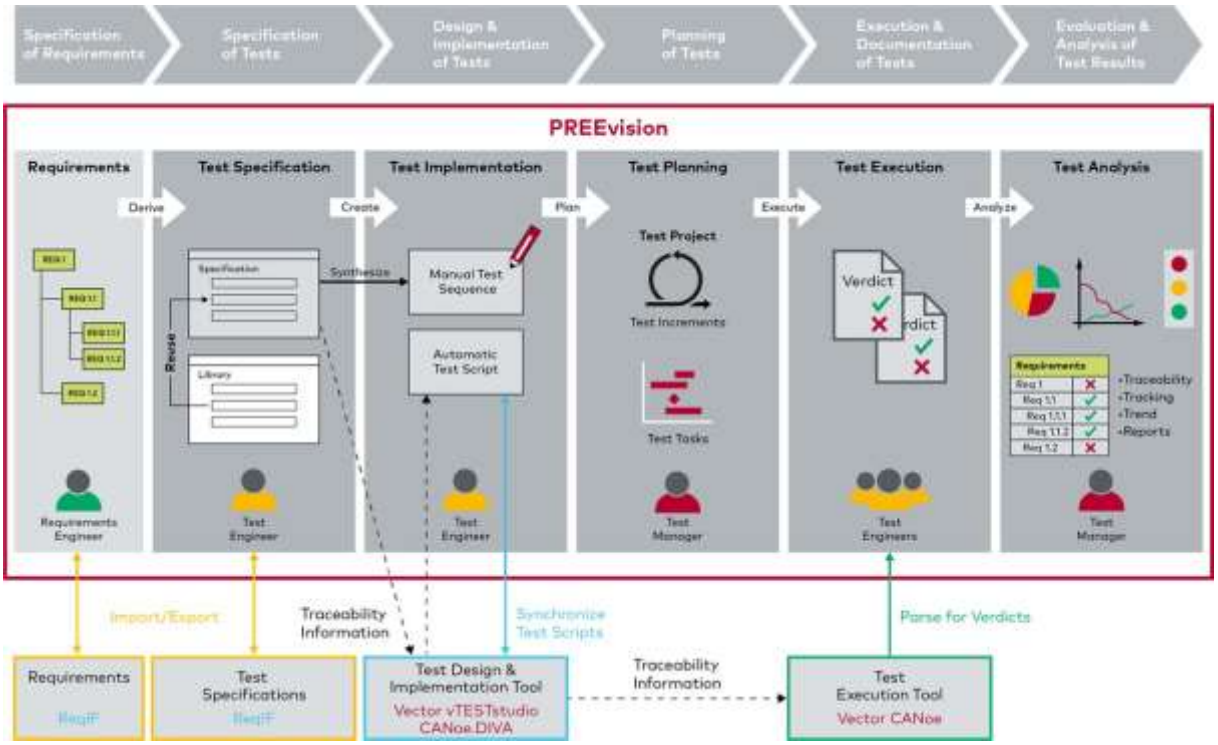

Figure 4-7: Test engineering and management in PREEvision

4.2 Vector TA Tool Suite

The Vector TA Tool Suite includes the Options TA.Design, TA.Simulation, and TA.Inspection which were developed exclusively for embedded multi- and many-core processor systems for the evaluation and optimization of real-time, performance and reliability properties applied by Project Managers, Software Architects, Software Developers, Integration and Test Engineers alike.

With it, users model embedded real-time systems with the graphical model editor or import existing systems from standard interfaces (e. g. AUTOSAR, AMALHTEA) as well as from hardware target traces. The models are used to simulate different system design alternatives at early stages in the development process. A comparison of software implementation alternatives and, consequently, their optimization regarding real-time and performance properties is facilitated. It enables well-founded forecasts about the system utilization and dimensioning of the hardware. An exemplary use case is to import hardware target traces of the software, automatically re-construct a timing model from it, analyse the target trace regarding real-time and performance properties, and compare this trace to simulation results.

In the context of XANDAR, the timing simulations features prominently. The timing simulation in the Vector TA Tool Suite is model-based and probabilistic. To achieve a suitable level of detail and accuracy – for XANDAR and beyond – the model contains three baseline parts: A hardware, operating system, and software model. The first describes the hardware platform used, e. g. a generic tri-core processor. The seconds includes operating system specifics, such as overheads (e. g. for interrupts) or scheduler details. The software model contains model information for all layers above the operating system, including basic software stacks and the actual application software. Figure 4-8 shows a graphical representation of the software model mapped to the operating system runtime model. A periodically activated task (container of

executable pieces of software in the real-time OS) contains pieces of the software model. It includes a decision tree based on a signal value for which a call sequence consisting of a runnable (executable piece of software) call and the setting of an operating system event.

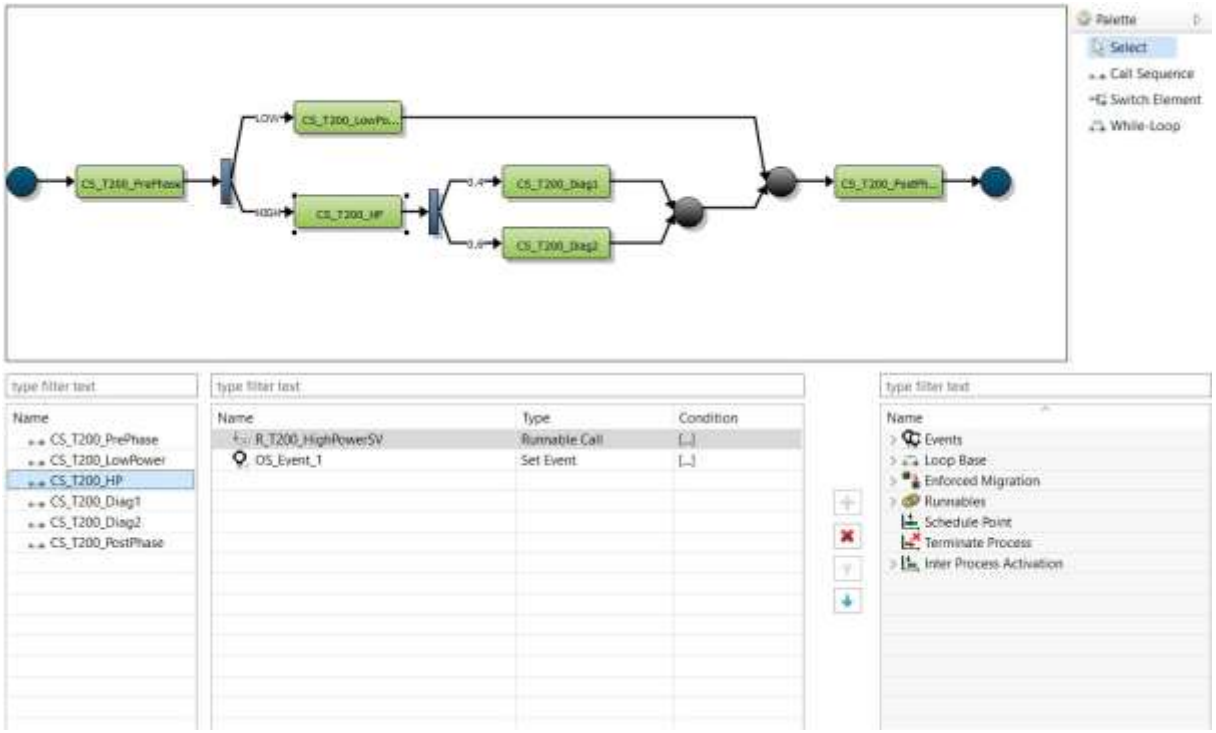

Figure 4-8: Vector TA Tool Suite: Graphical view of the timing simulation software model

To support the "x-by-construction" approach in XANDAR, the definition and verification of requirements is required. The Vector TA Tool Suite enables this workflow with regards to timing requirements and constraints, and consequently is used here in terms of "timing-by-design". To facilitate this, the model also features timing requirements. The tool itself then allows the users to verify these timing requirements either by simulation or by analysing a trace from a hardware target. The timing requirements in the Vector TA Tool Suite are partly compatible with the timing requirements/constraints in AMALTHEA and the Timing Extensions in AUTOSAR. However, as has been outlined in the project proposal, extensions are expected to be required to enable early timing verification on system level, as prescribed by the timingby-design paradigm.

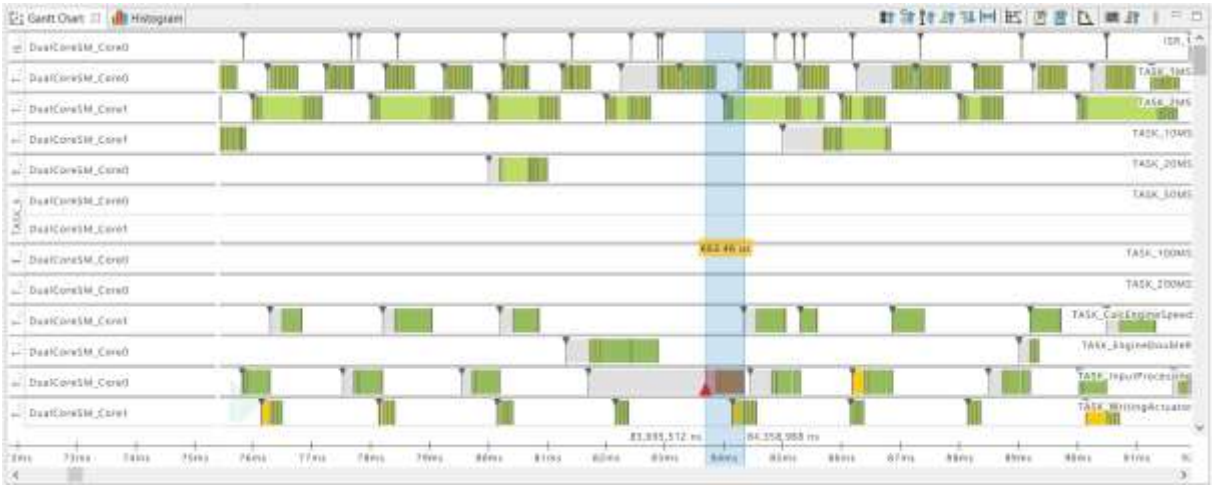

Figure 4-9: Vector TA Tool Suite: A Gantt chart view of a timing simulation

A visual representation of the results of a timing simulation that includes (potential) timing requirement violations is the Gantt chart in the Vector TA Tool Suite, as shown in Figure 4-9. In combination with a timing report that also details violations, this view specifically provides the user with actionable insight into timing requirement violations. Depending on the kind of requirement and context of the violation, the users can identify potential changes to the design of the system to improve its performance. In the example above, the Task "InputProcessing" has a response time that violates the deadline by about 660µs.

4.3 Ptolemy II

Ptolemy II is an open-source software framework [3] developed as part of the Ptolemy Project, which in turn is a research effort conducted by the Industrial Cyber-Physical Systems Center (iCyPhy) at UC Berkeley. The framework is developed in the Java programming language and focuses on the heterogeneous modelling of complex systems. In particular, it is concerned with cyber-physical systems (CPS), a class of systems characterized by a tight integration of computation and physical processes [4].

From an engineering perspective, the development of a system exhibiting such a tight integration is often complicated by the fact that a variety of modelling techniques must be combined to capture all the aspects that are relevant during the development process. While a physical plant, for instance, might be described using a set of ordinary differential equations (ODE), the operating states of an embedded component will more likely be captured using a suitable finite-state machine (FSM). As a further example, consider a digital filter from the signal processing domain. The exact dataflow within such components is usually described using a transfer function, a difference equation, or an implementation schematic that consists of delay blocks, adders, and scale blocks.

To tackle the heterogeneity involved in the modelling of such systems, Ptolemy II introduces the concept of an *actor*. Actors are executed concurrently and communicate with other actors by writing messages to their outputs ports as well as reading messages form their input ports [1]. The key to heterogeneous modelling in Ptolemy II is that the framework itself specifies only the abstract semantics of interactions between actors [3]. How exactly a specific interaction

takes place is described by the *domain* that is used in the potion of the model that contains the communicating actors. A domain is a specific implementation of a *model of computation*. The model of computation defines the concrete semantics that determine the exact interactions. When constructing a Ptolemy II model, the domain that is used within a certain model portion is specified by choosing the *director* that corresponds to the desired domain and incorporating this director into the model.

As an example, consider the Ptolemy II actor model shown in Figure 4-10. This model employs the synchronous dataflow (SDF) director, i.e., executes the model in the SDF domain. The model itself consists of two actors called "Ramp" and "Display", respectively. The only output port of the Ramp actor is connected to the only input port of the Display actor. This means that all messages generated by the Ramp actor are presented to the user via a console. Such a message is represented by a token that is generated by the Ramp and consumed by the Display actor. For the specific case of the SDF domain, in every iteration, a fixed number of tokens is produced by every output port and a fixed number of tokens is consumed by every input port. In the simple example shown in Figure 4-10, one token is generated by the Ramp and one token is consumed by the Display during every iteration. Executing five iterations of this model does therefore lead to the Display output shown in Figure 4-11.

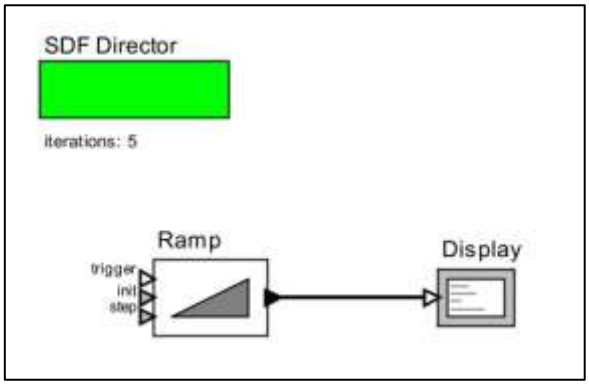

Figure 4-10: Ptolemy II actor model with two actors in the SDF domain

Figure 4-11: Output generated by the Display block in the SDF domain

The refined model shown in Figure 4-12 employs a "SequenceToArray" actor to repeatedly create a two-element array from every pair of tokens. Such a two-element array will then act as a new token that is forwarded to the Display port. In this specific case, the SDF scheduler determines that the Ramp must fire twice during every iteration. This means that two tokens are generated by the Ramp and consumed by the SequenceToArray actor during every iteration. At same time, the SequenceToArray actor produces exactly one token during every iteration. As a consequence, the Display outputs the value of "{0, 1}" during the first iteration.

In the second iteration, it outputs a value of "{2, 3}". A model in which the number of firings per iteration differs between iterations is referred to as a *multirate* model [1].

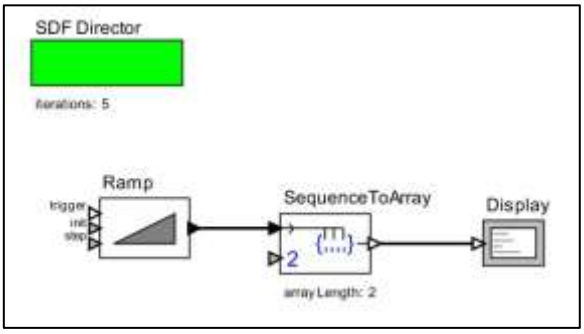

Figure 4-12: Refined actor model creating a two-element array from every pair of tokens

Ptolemy II provides various further directors that can be used during the construction of a model. Each of the corresponding domains exhibits a specific behaviour with respect to the way in which actors exchange messages. The interaction between actors in the SDF domain, which was outlined above, is example of such a communication behaviour.

A comprehensive description of the most common domains can be found in [1]. For the purposes of this document, the following list gives an overview of some of these domains:

1. **Synchronous dataflow (SDF):** Messages between actors are exchanged using FIFO queues. The number of times a specific actor fires during an iteration is fixed throughout the execution of the model and can be statically determined. This domain is usually untimed, i.e., does not have a notion of time.
2. **Dynamic dataflow (DDF):** Similar to SDF, but the exact communication patterns do not have to statically computable. This domain allows users to make use actors that consume tokens from their inputs in a selective manner. It is usually untimed.
3. **Process network (PN):** Actors execute in parallel threads and communicate using asynchronous message passing. This means that writing to an output port corresponds to a non-blocking writes to a FIFO queue. Usually untimed.
4. **Finite-state machine (FSM):** A special domain used to model finite-state machines. In contrast to all other domains mentioned in this document, this domain makes use of so-called "bubble-and-arc diagrams" instead of block diagrams.
5. **Discrete event (DE):** Actors produce events that have a value and a time.
6. **Continuous (CT):** To describe the time behaviour of continuous physical processes.

It is important to understand that the mentioned six domains are only selected examples of the domains that Ptolemy II provides. Furthermore, the software framework is extensible in the sense that custom domains can be implemented using Java.

After describing the general concept of a domain and providing a list of some commonly used ones, it is now possible to describe how Ptolemy II allows users to model heterogeneous systems by integrating various domains into one model. The key to this capability is to understand that Ptolemy II models can be arbitrarily nested to create a hierarchical model. An actor used from the DE domain, for instance, can be a "sub-model" that itself obeys to the SDF rules. Using the ports of the actor, the designer is able to interconnect SDF actors instantiated from the sub-model and DE actors that are on the hierarchy level as the sub-model itself.

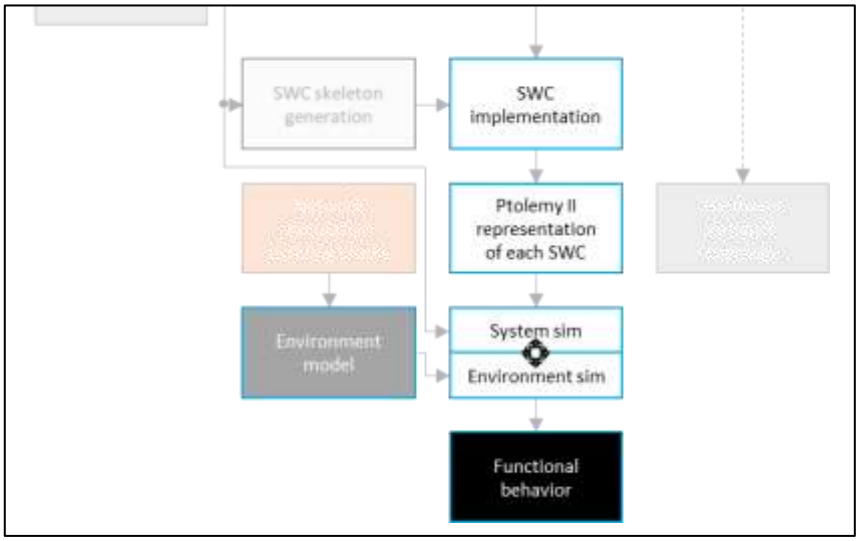

Figure 4-13: Functional verification path of the XANDAR development process

As described in section 2.2.3 of this document, the envisaged XANDAR development process makes use of Ptolemy II for the model-based verification of the functional behaviour. Figure 4-13 shows the relevant excerpt from the visualization of the overall development process (see Figure 2-1) and highlights steps related to functional verification in blue.

More specifically, the process expects the developer to model the environment in which the interconnected SWCs are planned to be deployed using Ptolemy II. Depending on the exact nature of the functional requirements to verify, this environment model will make use of a specific combination of Ptolemy II domains. If in a particular application scenario, for instance, it is necessary to verify that the developed SWCs respond a given input sequence with the corresponding output sequence, it might be sufficient to describe the environment using the SDF director. In another example scenario, it might be necessary to verify that the response time exhibited by the SWC network falls into a specific range. In this case, a DE or even a CT model might be suitable choices for the environment model.

In any case, the XANDAR development process is based on the general idea that a suitable representation of every SWC implementation is incorporated into the Ptolemy II simulation. To do so, every SWC will need to be made available as an actor in the Ptolemy II framework. The specific domains in which such an actor has to be made available depends on the requirements established by the use cases (see D1.1 of the XANDAR project). Furthermore, the exact mechanisms in which a specific SWC implementation is adapted to these domains is highly dependent on the characteristics of the RTE mechanisms that the XANDAR toolchain will support in the future. Both aspects will be analysed in more detail and answered in the future work of WP2, WP3, and WP4.

4.4 Network Simulator 3 (NS-3)

NS-3 is a discrete event network simulator, developed to provide an open, extensible network simulation platform for networking research and education. The NS-3 software infrastructure encourages the development of simulation models which are realistic, allowing NS-3 to be used as a real time network simulator, with real world interconnection. One of the key features

of NS-3 is that designed as a set of libraries that can be combined with external software libraries, such as Linux OS libraries. Finally, the development of models in NS-3 made with C++ and Python scripts. Figure 4-14 below shows a basic simulation model in NS-3.

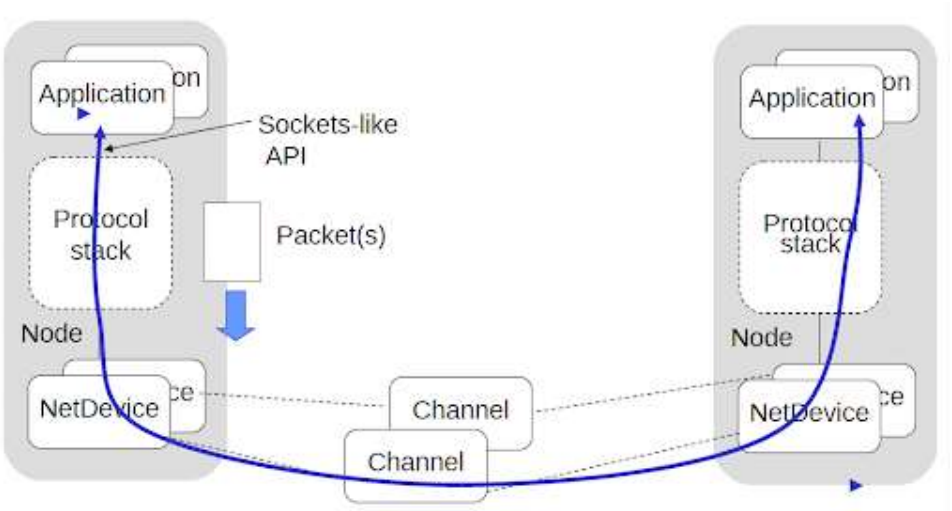

Figure 4-14: Simulation model in NS-3

In NS-3 the basic computing device abstraction is called the Node. Node class provides methods for managing the representations of computing devices in simulations, with added functionality. NS-3 node operates similar to applications, protocol stacks and peripheral cards with their associated drivers.

The basic abstraction for a user program that generates some activity to be simulated is the application, represented by the Application class. The Application class provides methods for managing the representations of user-level applications in simulations.

The net-device abstraction, represents a peripheral hardware device. Net-Devices can be attached in a node, allowing node to communicate with each other in the simulation via Channels. The net-device abstraction covers both software driver and the simulated hardware. Finally, a node may be connected to more than on Channel via multiple NetDevices.

Channels in NS-3 represents basic communication. The Channel class provides methods for managing communication subnetwork objects and connecting nodes. A Channel's modelling capabilities vary from simple wire networks to more complicated networks like large Ethernet switches, or wireless networks in three-dimensional space full of obstructions.

Network topologies in NS-3 can be built by parsing traces from topology mapping engines or user created network generator from NS-3 simulator. Known topology traces supported by NS3 are:

- Orbis 0.7 traces,
- Inet 3.0 traces, • Rocketfuel traces.

Mobility of nodes is a factor that introduces significant complexity in the study and evaluation of network configurations and operation. NS-3 support mobility models, that include:

- set of mobility models which are used to track and maintain the current Cartesian position and speed of an object. NS-3 uses only the Cartesian coordinate system;

- a "course change notifier" trace source which can be used to register listeners to the course changes of a mobility model;
- a number of helper classes which are used to place nodes and setup mobility models (including parsers for some mobility definition formats).

Energy consumption is a key issue for wireless devices, and especially for wireless networks. The NS-3 Energy Framework provides the basis for energy consumption, energy source and energy harvesting modelling. The NS-3 Energy Framework is composed of three parts, listed below:

Energy Source: The Energy Source represents the power supply on each node. A node can have one or more energy sources, and each energy source can be connected to multiple device energy models.

Device Energy model: The Device Energy Model is the energy consumption model of a device installed on the node. It is designed to be a state-based model where each device is assumed to have a number of states, and each state is associated with a power consumption value.

Energy Harvester: The energy harvester represents the elements that harvest energy from the environment and recharge the Energy Source to which it is connected. The energy harvester includes the complete implementation of the actual energy harvesting device (e.g., a solar panel) and the environment (e.g., the solar radiation).

5 Existing data models

5.1 PREEvision data model

For modelling the entirety of electric/electronic systems, from requirements to software and hardware architectures to the wiring harness, PREEvision provides a comprehensive data model with dedicated abstraction layers [2].

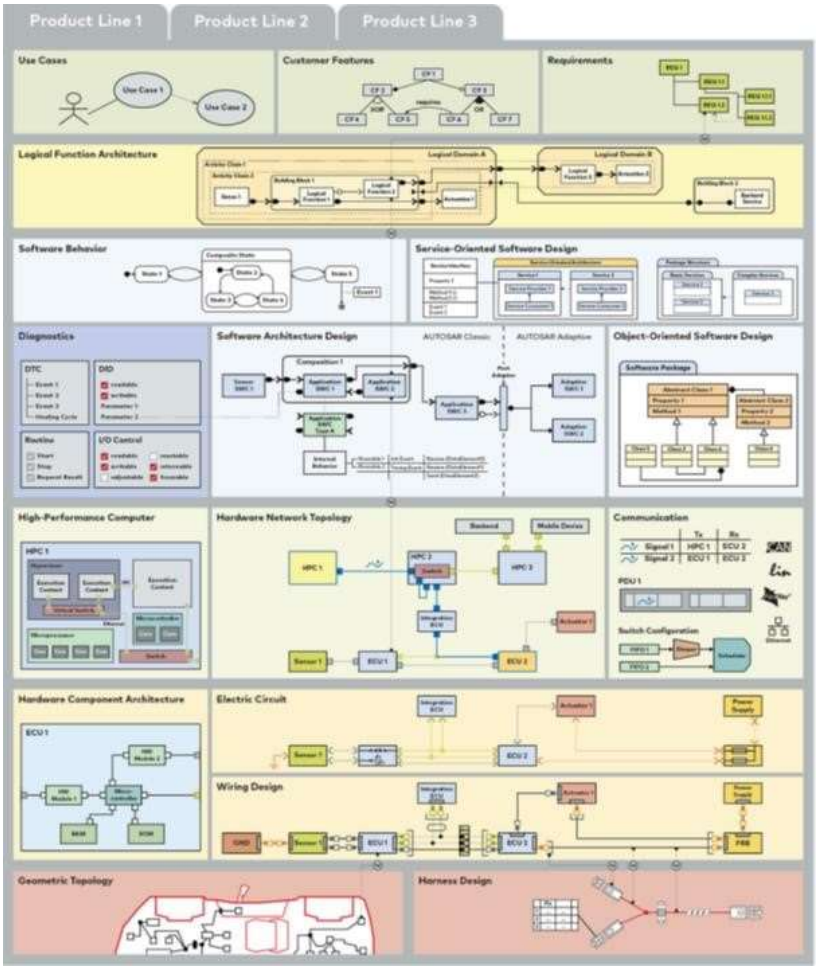

Figure 5-1: Scope of the PREEvision data model

Requirements: System designers develop use cases, customer features, and corresponding requirements. Via links and mappings, the implementation of customer features and requirements can be traced at any time.

Logical Function Architecture: In the logical function architecture, the requirements are implemented by logical functions that are connected via ports and interfaces. The resulting function network is the basis for the technical development of hardware and software.

Software/Service Architecture: In the system and software architecture, software components, their behaviour, and their interfaces are modelled. A service-oriented modelling is possible. The software architecture with libraries supports the AUTOSAR method. Additionally, all implementation artefacts can be managed.

Diagnostics: On the diagnostics layer, diagnostic objects can be described and linked to the application software. This ensures consistency between diagnostics and its realization in software.

Hardware Architecture: On the hardware layer, ECUs, high-performance computer, sensors, and actuators, their networking via bus systems as well as the power supply are modelled.

Communication: On the communication layer, one defines how software components exchange data across hardware borders. PREEvision supports all relevant network technologies including CAN, CAN FD, LIN, FlexRay and Ethernet.

Electric Circuit and Wiring Design: In the electric circuit diagram, the electric characteristics of the components and their interconnections are defined. Also, the internal electrical design of components with fuses and resistors can be modelled. On the wiring harness layer, one defines the physical details of the wiring harness including pins, connectors, cables, inline connectors, and splices.

Geometry and Harness Design: In the vehicle geometry, installation spaces and locations are defined or imported via 3D KBL data. Then routing paths via topology segments including inline connectors are modelled.

5.2 TA Tool Suite data model

The RTE model (Real-Time Evaluation) describes all dynamic properties of a real-time multicore system. It is used for metric calculation, discrete system simulation, and optimization. The RTE model is divided into the following sections:

- Hardware Model: Describes hardware elements like ECUs, processors, and memory elements.
- Software Model: Describes the application software architecture, e.g. tasks, ISRs, and functions.
- Operating-System Model: Describes scheduling units and the services provided to the application software like semaphores and buffering.
- Stimulation Model: Describes the behaviour of the environment which effects the execution of the software, e.g. activation of tasks or ISRs.
- Mapping Model: Describes the allocation of tasks or ISRs to scheduling units as well as the mapping of stimuli to tasks or ISRs.

The RTE model in general is generated by an AUTOSAR description or a system description database. Using this model, it is possible to execute a discrete event simulation to analyse the timing behaviour, efficiency, and robustness. Based on the trace of the system, different metrics like response time or net execution time could be derived. Afterwards a metric evaluation can be done, by statistical views of the determined metrics. The RTE model is used furthermore for a model analysis, e.g., data flow between runnables. Additionally, the simulation generates a trace, which can be evaluated afterwards. Moreover, it is possible to use the RTE model for model-based optimization. Starting with an initial/or no allocation of tasks to cores, the optimization tool automatically improves system properties.

5.3 EAST-ADL

The original EAST-ADL was developed from 2000 – 2004 in the ITEA project EAST-EEA, see [5]. The main goal of the project was to define an open system architecture, build an appropriate integration platform and establish a standard for the European and worldwide automotive market.

The standard was supposed to ensure the interoperability of hardware and software and thus improve the reusability through component–based design. With this the developers hoped to minimize costs and strengthen their leading position within the automotive market. A part of this project was the definition of the EAST-ADL, an architecture description language (ADL) to support the development of embedded software for the automotive domain.

In 2006 the ATESST project started and continued the development of EAST-ADL, aligning it with the then–emerging AUTOSAR standard (see Section 5.4). The result was EAST-ADL2 [6], a revised version of the original EAST-ADL. The lower levels of the original EAST-ADL were modified and became part of AUTOSAR standard. These lower levels were replaced by the Implementation Architecture which represents the actual link/transition to AUTOSAR. Furthermore, an UML2 [7] profile of EAST-ADL2 was defined to seamlessly integrate it with UML modelling tools like Rhapsody. The system of abstraction layers of EAST-ADL2 and AUTOSAR is displayed in Figure 5-2.

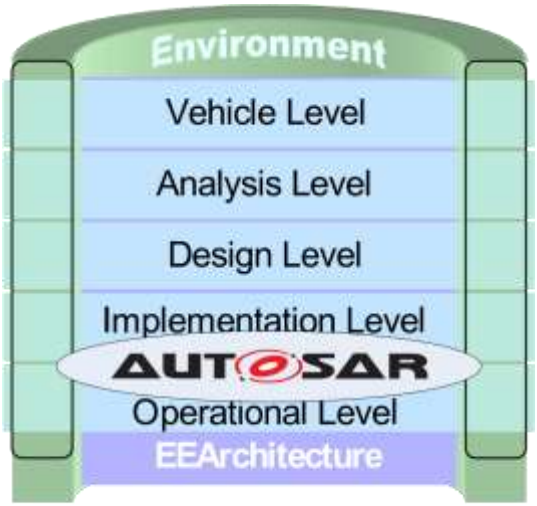

Figure 5-2: Abstraction layers in EAST-ADL2

As depicted, the EAST-ADL2 model is structured in different abstraction levels (i.e., Vehicle, Analysis, Design, and Implementation level) which map to the abstraction levels given in the automotive safety standard ISO 26262 [8]. Additionally, the EAST-ADL2 model covers a variety of orthogonal concerns, like requirements, structure, timing, dependability, and several more. We will give an overview of all these packages in the following.

The *Structure* package defines the static structure of the instances of the system being modelled and their static relationships. This includes the instance's internal structure as well as its external interfaces. Within this package, each abstraction level has a corresponding model artefact as follows:

- Vehicle Level includes the Technical Feature Model.

- Analysis Level includes the Functional Analysis Architecture (FAA).
- Design Level includes the Functional Design Architecture (FDA), the Hardware Design Architecture and the Allocation.
- Implementation level includes a reference to an AUTOSAR model.

The Structure package is UML2 [7] compliant.

The *Environment* package describes the environment of the vehicle electric and electronic architecture by means of continuous functions. The *Behavior* package describes either a function performing some computation on provided data, or the execution of a service called by another function. Within EAST-ADL2, the execution of the behaviour assumes a strict runto-completion, single buffer-overwrite management of data. Also, the execution is nonconcurrent within an elementary function. The *Variability* package provides means to express variability in the FAA, FDA, and implementation level architecture.

The *Requirement* package describes conditions or capabilities that must be met or possessed by a system or a set of components to satisfy a set of (formally imposed) properties. EASTADL2 offers modelling artefacts for dealing with the highly changing nature of requirements, e.g., courses, types, levels of abstraction, stakeholders, etc. The Requirement package, like the Structure package, is UML2 compliant. With the *Timing* package, timing constraints according to TADL2 [9] can be specified. The *Dependability* package supports:

- the definition and classification of safety requirements via a preliminary Hazard Analysis Risk Assessment,
- tracing and categorising safety requirements according to their role in the safety life cycle,
- formalising safety requirements using safety constraints,
- formalising and assessing fault propagation through error models, and
- organising evidence of safety in a Safety Case

This package is designed to support the automotive standard for Functional Safety, ISO 26262.

Finally, the *GenericConstraints* and *Infrastructure* packages are utilized for the specification of properties, requirements, or validation results, for identified elements, and for the specification of the infrastructure constructs, respectively. It is worth noting that, although EAST-ADL2 is aligned with AUTOSAR, it does not play a major role as an industrially used data exchange model. Commercial tools mostly use other exchange formats like AUTOSAR, UML, SysML [10], ReqIF [11], etc. To sum up, EAST-ADL2 may be suited for system level modelling, however, it might not be supported by commercial tools as an interface for data exchange.

5.4 AUTOSAR

AUTOSAR (AUTomotive Open System ARchitecture) [12] is a standard (process/data exchange) developed and mainly used by the automotive industry. It was initiated in autumn 2003 by the need to manage the growing software complexity in electric/electronic systems. Enabling software component reuse, variability, accelerated development, cost optimization, system scalability, etc., it defines standardized highly configurable interfaces and design processes. A simplified version of that design process is depicted in Figure 5-3.

Generally, AUTOSAR can be used after the initial system design (when it is clear which parts will be software and which will be hardware), i.e., AUTOSAR is targeted for software design and configuration. As a starting point for the AUTOSAR process there are some system constraints to initiate the system configuration and software component (SWC) description. Since the system configuration step relies on a SWC description the latter step is done first. Then, during the system configuration, the SWCs are deployed to computing resources (Electronic Control Units - ECUs) and the communication between them is defined. After that, the SWCs can be implemented in parallel to the ECU configurations. The system configuration is usually done by Original Equipment Manufacturers (OEMs), while a single ECU is provided by suppliers. The interface (subject matter of the contract) between OEM and suppliers is the ECU Extract for the supplier specific ECU. It contains only the communication relevant for that ECU and its SWCs. Suppliers can now extend this ECU Extract by configuring it. During the ECU configuration additional SWCs may be introduced by the supplier.

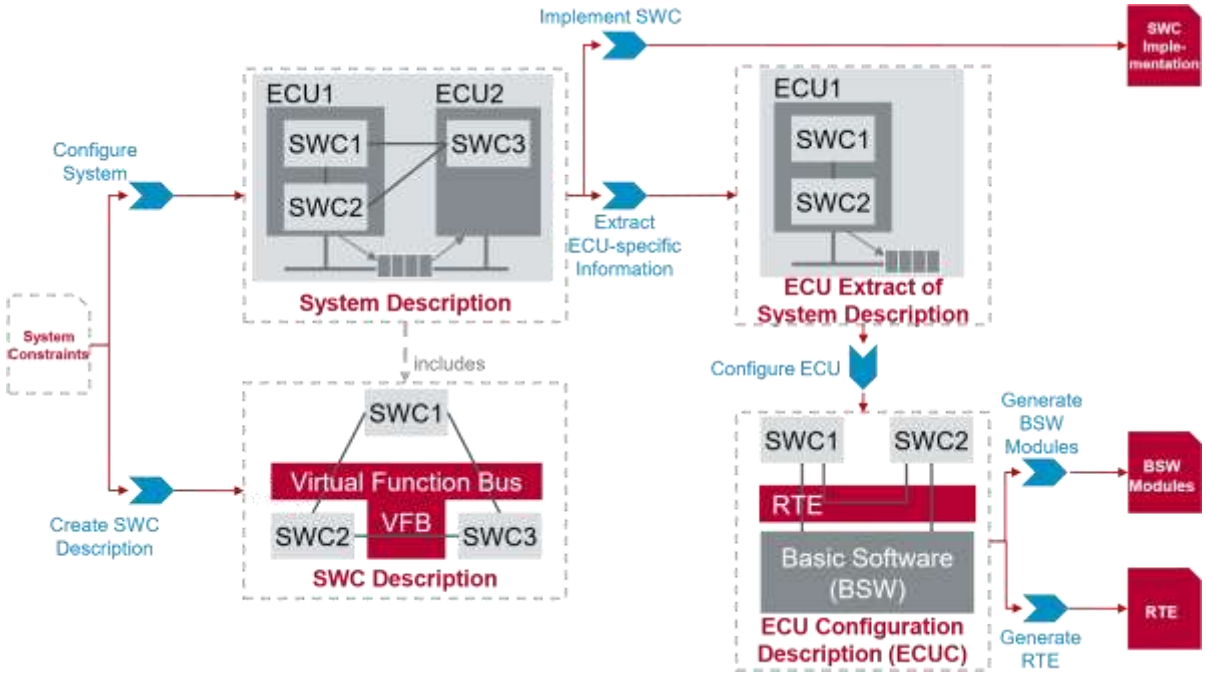

Figure 5-3: Simplified AUTOSAR Classic Platform development process

SWCs communicate via a standardized interface, the Virtual Function Bus (VFB). Refined interfaces of the VFB and their implementations are called Runtime Environment (RTE), i.e., the RTE is the realization of the VFB. On the lower abstraction level of the hardware, there is the Basic Software (BSW) which takes care of hardware specifics, e.g., Input/Output, low level drivers, etc. In addition to the BSW there is the operating system (OS) which governs the resource sharing (computation time and physical resources like memory) among the SWCs. Once the ECU is configured, the implementation of the BSW modules, the RTE, and the OS can be generated. Combined with the SWC implementation, these artefacts can be compiled to a binary which can be flashed to the target device for execution/testing.

There are several meta-model parts and their extensions (e.g., TimingExtensions) in AUTOSAR. As indicated in Figure 5-3, there are two layers/views of interest for AUTOSAR models: System and ECU layer. On the system layer, the general communication between ECUs is important to deploy functionality (SWCs) to appropriate computation nodes. Hardware resources, SWCs, and communication signals are modelled here. On the ECU layer, many

details can be described, it is also possible to model new parameters and assign values to them. More details about hardware resources, operating system specifics, communication protocols, partitions, etc. can be modelled here. Most of these details are modelled via parameter and value definitions.

On top of the general two layers there are crosscutting artefacts like constraints, data types, mappings, and even SWCs (they are used on both layers). SWCs can be decomposed into subcomponents, constituting an arbitrarily deep compositional hierarchy. Leaf SWCs (not further decomposed) have Runnable Entities (REs) which can be combined to OS tasks (different composition) to be scheduled by an OS for efficient resource sharing. With AUTOSAR all these artefacts can be modelled, persisted, and exchanged in well-defined AUTOSAR XML files.

In recent years, the demand for computational performance in AUTOSAR systems has been increasing rapidly. To address this, the AUTOSAR adaptive platform (AP) was introduced. While the AUTOSAR Classic Platform (CP – described in the above overview) is targeted for highly configurable static systems (the configuration and implementation are transformed to one monolithic binary which cannot easily be updated, once delivered), the AP addresses high performance hardware and offers more flexibility – application binaries can be updated without changing the rest of the system. The AP thus provides less configuration possibilities since these are now mostly done in the implementation. On the one hand, this leaves more flexibility during the runtime of the system, and even an AppStore-based dynamic software configuration with over-the-air updates could be realized (not planned in XANDAR). As a result, on the other hand, model-based analyses are rather limited without considering the source code. So, the AP comes with some uncertainties for the validation and verification of the system.

The development process for adaptive AUTOSAR systems is very similar to the CP. The main difference is that in the AP the ECU configuration part is simpler – it relies on a POSIX compliant operating system to manage the underlying hardware resources. On top of this, the AP provides a standardized runtime API for applications. Like CP, these APIs and configuration artefacts are described in several parts (templates) of the standard.

In the following subsections, we will describe the most relevant AUTOSAR CP and AP standard parts. Since the standard contains tens of thousands of pages, we cannot be extensive here. Some parts are specific to either the CP or AP while others are more abstract.

5.4.1 AUTOSAR Classic Platform

5.4.1.1 ECU Resource Template

The ECU Resource Template [13] provides possibilities for the description of the hardware composition of an Electronic Control Unit (ECU)). The template however is just used to describe the hardware resources. Instances of those are then used elsewhere for example to define the topology of a particular system [10]. This is the reason, why other templates have references to the ECU Resource. Modelling elements according to the ECU Resource Template is done hierarchically. That way, different levels of detail for the description of the hardware are supported. A particular ECU can, for example, be described as a hierarchical composition of micro-controllers and peripheral electronics.

5.4.1.2 Software Component Template

The Software Component Template [14] provides possibilities for the definition of applications, which are formally described pieces of software. Therefore, the following three distinctive levels of abstraction are available [15]:

- **Structure**: To support the re-usability of application software AUTOSAR uses the socalled prototype pattern for the structural description. This concept allows to create hierarchical structures of software components with arbitrary complexity. In AUTOSAR every software component can either be an atomic software component or a composition. While atomic software components cannot be further decomposed, compositions represent in turn a hierarchical structure of software components again, which can be distributed across multiple ECUs. Thus, an atomic software component encapsulates the actual functionality of a piece of software. Furthermore, there are several different types of software components defined in AUTOSAR to classify them according their specific field of application. This part of the Software Component Template also describes the fundamental communication properties of those components and their communication relationships among each other. In AUTOSAR components can only communicate with each other via ports. Thereby different communication paradigms like sender-receiver or client-server communication are supported. Furthermore, each of these ports is defined by an interface.
- **Behaviour**: As mentioned above an atomic software component encapsulates the actual functionality of a piece of software. The behavioural section provides possibilities to formally define their behaviour. Furthermore, AUTOSAR supports the concept of Runnable Entities. According to the AUTOSAR glossary a 'Runnable Entity [16]. They are, thus, the smallest code fragments that can be subjected for scheduling by the underlying operating system. The execution of a runnable entity is always triggered by a RTE event. Runtime Environment (RTE) events in turn will occur during the execution due to different reasons, such as the reception or dispatch of data, the request of an operation or just the periodic invocation of a runnable. Furthermore, the internal behaviour describes the ongoing communication among Runnable Entities during runtime. AUTOSAR supports therefore several communication concepts like sender/receiver or
- **Implementation**: Finally, on the lowest level of abstraction the actual implementation is described. It not only references the source code or generated object code artefacts but also allows to provide information about the resource consumption and the tools necessary for a certain software release

5.4.1.3 BSW Module Description Template

The BSW Module Description Template [17] provides, like the Software Component Template, possibilities to formally describe pieces of software. This template however focuses on the definition of Basic Software (BSW) artefacts instead of software applications. According to the glossary of the AUTOSAR specification 'Basic Software provides the infrastructural [...] functionalities of an ECU' [16]. Examples for these basic functionalities are, for example, communication layers, services, and drivers.

Their thematic similarity results also in a common structural resemblance. Thus, the BSW Module Description Template, like the Software Component Template, knows three distinctive levels of abstraction. However, while the whole structure of a software application must be specified, for BSW modules it is just necessary to define a single interface, which represents the entry for its execution. Both other focuses of the Software Component Template, the description of the internal behaviour and the implementation details, are also an important part of the BSW Module Description Template. The concept of the definition thereby remains the same, just the names of the elements change. So, for example, Runnable Entities become Schedulable Entities in the context of BSW Modules and RTE Events become BSW Events. For that reason, a more detailed description of the BSW Module Description Template is abandoned at this point and Section 5.4.1.2 above and [17] referenced instead.

5.4.1.4 System Template

The System Template [18] defines relationships between the software architecture and the hardware structure of the whole system. It, thus, brings the software view on the system, defined by the software component compositions, together with the physical view, which is represented by networked ECU instances.

This document is used to create the System Constraint Description as well as the System Configuration Description. Latter specifies a set of constraints, which must be considered during the system configuration, while former describes the system under development. Therefore, the System Template defines the following five major elements:

- **Topology**: This part makes it possible to model the physical system topology of a vehicle in AUTOSAR. The topology is thereby determined by a set of ECU instances according to the definitions in the ECU Resource Description and the buses, which connect those instances to communication clusters.
- **Software**: One of the most important tasks of the System Template is to provide a way to define the communication capabilities and the connections between software components. This is done in a hierarchical structure of software components such that a given system is represented just by one top-level-composition, which in turn can then contain more software components.
- **Communication**: This element covers constraints or configurations that describe the information exchange between the ECUs. To determine such communication aspects, AUTOSAR uses Field Bus Exchange Format (FIBEX) elements. The information on which signals, frames or PDUs are sent and received on which channel at the single ECU is then stored within a communication matrix.
- **Mapping**: The fourth part is the system mapping. It holds a set of mappings like the mapping of software components to an ECU, the mapping of a software component to an implementation, the mapping of an ECU hardware type to an ECU instance used in a physical topology. Furthermore, it is used to map operations and data elements used in the ports of the software components to signals.
- **Mapping Constraints**: The last major topic covered by the System Template is the possibility to constrain the mapping of software components. It thus allows to define invariants that must be fulfilled by a valid mapping. Therefore, it supports constraints

that on the one hand express which software components must be mapped together on the same ECU and on the other hand which must not be mapped to the same ECU.

5.4.1.5 ECUC Description Template

The ECU Configuration Description Template [19] covers two different subjects. On the one hand it specifies a way to declare parameters and their permitted occurrences for the configuration of a certain software module. This is covered by the ECU Configuration Definition meta-model. On the other hand, it provides by the ECU Configuration Value description a way how to equip these parameters with actual values to define a configuration for generating a working executable for a single ECU.

- **ECU Configuration Definition**: EcucModuleDef denotes the definition of parameters for a certain software module while EcucContainerDef is used to gather common attributes. The actual parameter definition is finally described hierarchically using the ECU configuration parameter configuration container. Therefore, a set of parameter types is available to express the different data types of a parameter (e.g., Boolean, Integer, Float, String).
- **ECU Configuration Value**: The value description is done correspondingly to the parameter definition. EcucModuleConfigurationValues denotes the configuration of a certain software module, EcucContainerValue contains the common attributes and EcucParameterValue represents a specific attribute. Each configuration element references its corresponding definition. That way the individual parameter types don't have to be distinguished in the ECU Configuration Value any more since their type is determined by the referenced definition.

5.4.2 Adaptive Platform

Traditionally, ECUs mainly implement functionality that replaces or augments electromechanical systems. Software in those deeply embedded ECUs controls electrical output signals based on input signals and information from other ECUs connected to the vehicle network. Much of the control software is designed and implemented for the target vehicle and does not change significantly during vehicle lifetime.

New vehicle functions, such as highly automated driving, will introduce highly complex and computing resource demanding software into the vehicles and must fulfil strict integrity and security requirements. Such software realizes functions, such as environment perception and behaviour planning, and integrates the vehicle into external backend and infrastructure systems. The vehicle's software needs to be updated during its lifecycle, due to evolving external systems or improved functionality.

The AUTOSAR Classic Platform, as described in Section 5.4.1, addresses the needs of deeply embedded ECUs, while it cannot fulfil the needs of the ECUs described above. Therefore, the AUTOSAR Adaptive Platform (AP) provides mainly high-performance computing and communication mechanisms and offers flexible software configuration, e.g., to support software update over-the-air. Features specifically defined for the CP, such as access to electrical signals and automotive specific bus systems, can be integrated into the AP but are not in the focus.

Adaptive Applications (AA) run on top of ARA – AUTOSAR Runtime for Adaptive applications. ARA consists of application interfaces provided by Functional Clusters. Any AA can also provide Services to other AA, called Non-platform services.

The interfaces of the Functional Clusters provide specified C++ interfaces, or any other language bindings AP may support in future. Note that underneath the ARA interface, including the libraries of ARA invoked in the AA contexts, the implementation may use other interfaces than ARA to realise the specification of AP. It is up to the provider of the AP to design the implementation.

The language binding of these API is based on C++, and the C++ Standard library is also available as part of ARA. Regarding the OS API, only the PSE51 interface, a single-process profile of the POSIX standard is available as part of ARA. PSE51 has been selected to offer portability for existing POSIX applications and to achieve freedom of interference among applications.

From the OS point of view, the AP and AA only constitute a set of processes, each containing one or multiple threads – there is no difference among these processes, though it is up to the implementation of the AP to offer any sort of partitioning. These processes do interact with each other through Inter Process Communication (IPC) or any other OS functionalities available. However, AA processes may not use IPC directly and can only communicate via ARA.

5.4.3 Timing Extensions

With AUTOSAR Release 4.0, the Timing Extensions found their way into the standard [20] [21]. Their purpose is to provide a consolidated and consistent way to represent relevant timing dependencies and constraints. This enables the analysis of a system's timing behaviour and the validation of the analysis results against timing constraints throughout the development process. Depending on the availability of necessary information in each development phase, the so-called views, different timing constraints are available to describe the timing behaviour of an AUTOSAR system:

- VfbTiming: This view deals with timing information related to the interaction of SwComponentTypes at VFB level.
- SwcTiming: This view deals with timing information related to the SwcInternalBehavior of AtomicSwComponentTypes.
- SystemTiming: This view deals with timing information related to a System, utilizing information about topology, software deployment, and signal mapping.
- BswModuleTiming and BswCompositionTiming: The first perspective deals with timing information related to the BswInternalBehavior of a single BswModuleDescription. Whereas the latter perspective deals with timing information related to BswInternalBehavior of more than one implementation of a BswModuleDescription.
- EcuTiming: This view deals with timing information related to the EcucValueCollection covering the entire software on an ECU with application software and configured basic software.
- MachineTiming: This view deals with timing information related to a Machine.

- ExecutableTiming: This view deals with timing information related to an Executable.
- ServiceTiming: This view deals with timing information related to a service, specifically AdaptivePlatformServiceInstance.

This can for example be the maximum repetition time between the starts of a Runnable Entity or the minimum distance between subsequent data accesses within a given time interval. Additionally, it is not only possible to specify particular occurrences of events but also sequence relationships between those during the runtime of a system by using Event Chains. That way it is possible to specify the timing behaviour of arbitrary scenarios within the whole system. A popular application is for example the specification of data flow from input over multiple Runnable Entities to output.

5.4.4 Abstract Platform

The abstract platform (XP) was introduced with AUTOSAR release R19-11 as part of the Adaptive Platform (AP) specification and was moved to the Foundation (FO) specification with release R20-11 [22]. It is intended for an early software design stage and allows one to model the interaction between the functional software blocks and specify the basics: i.e., signal names, the directional flow of the data (providers/consumers) and the physical data types. Further refinement of the design will be done in a downstream stage, thus, set the design and deployment on AP or CP or non-AUTOSAR platform and specifying what type of concrete component shall implement the function, or, which type of concrete interface provides the required data.

The specification aims to provide a system description of a functional model. It has its own system description to distinguish XP content from other types of system descriptions/extracts. The abstract platform formally describes the functional interactions on a component model level. The hierarchy of SWCs of arbitrary complexity is realized via the recursive relation of compositions that are defined in the component model. Each composition can have a set of PortPrototypes through which it can communicate. Via Connectors it is possible to indicate data dependencies, i.e., dedicated communication between individual ports is established.

5.5 AMALTHEA

5.5.1 Hardware description

The hardware model represents an abstract description of the hardware, especially of its internal procedures. In this description the modelled level of detail can vary from the number of processing cores and its processing performance up to a detailed memory topology and behaviour descriptions including memory modules, caches, bus networks, or crossbars. This information can, for example, be used in a timing simulation to improve accuracy of hardwaresoftware interactions.

5.5.2 Software description

Labels

The access to a label can either be read or write. Modelling the access to labels is important as it needs to be analysed regarding the memory structure. Starting from the labels, the accessing, runnables and services can be analysed if there is any concurrency existing.

Data types

The AMALTHEA data model supports base type definitions which are also often called primitive data types in the consensus of programming. It consists of the name and number of bits to define a data type. An additional information to store is the mapping information for a target platform. This type of information consists of how an own defined type is mapped to an existing data type. It is needed, as it could be different, like different keywords, or an own meta type for different platforms can be defined. The compound data types are data structures, based on given or defined base types. In the literature they are also often named composite or derived types. The result of this type of definition is an own data type, which can be used as base data types. They can consist of static structures or dynamic ones, like arrays or stacks.

The AMALTHEA data model supports the following compound data type definitions:

- *Pointer*: Holds a reference to another type using its memory address
- *Array*: Contains several elements of the same data type. The size of an array can be fixed or expandable.
- *Struct*: Contains other data types in a structured way, often called fields or members. The fields can be accessed by their name.

Memory Information

Analysing and mapping the software structure to available memories needs additional information of the included elements. This type of information targets the consumed size of memory of an element.

The element *AbstractElementMemoryInformation* is a generalized element for other elements to extend it in an object-oriented way. It contains a reference to the *DataSize* element, represented by the named size attribute. The *DataSize* contains the size in bits and provides convenience methods to set and get the size also in bytes. It is already internal converted, so the user does not need to care about it.

Section

The Section is an element to make a reference from existing software components to a target memory area. It includes references to the following objects:

- Labels
- Runnables
- Services

In addition, it contains a name and predefined information about the needed memory size. If the size is not explicit set, it must be computed based on the included elements. An algorithm can make a distribution based on sections or a developer can already define needed criteria for the target memory as input for the algorithm. This information is then included in the

mapping constraints. With this type of information available, the complexity of mapping software components to memories should be reduced.

Runnables and Services

Both elements, Runnables and Services, are an abstraction of an executable entity. Instead of tasks, which are providing a context for the operating system, Runnables and Services are including the instructions to perform. They include an abstraction of these instructions using different algorithms, based on performance data. The difference between Runnables and Services is based on their activation and type of calling. While the initial activation of a runnable can only be performed by a task or another runnable, services can only be activated by Runnables or other Services. Runnables do not have any parameters by calling them, as Services could have. They can be initialized having Labels or Services as parameter, using a reference or pointer to the target element. Based on this type of information, an additional analysis can be performed.

Activity Graph

The content that a *Runnable* or a *Process* is executing, is defined by a so called *AcitvityGraph*. It controls the flow of different calls.

The class *ActivityGraphItem* generalizes all kind of calls that can be executed. The call *RunnableCall* executes a referenced Runnable. *InterProcessTrigger* enforces a stimulus to activate its processes. *EnforcedMigration* lets the executing process migrate to the core which is controlled by the referenced scheduler. With a *SchedulePoint*, the scheduler the task is mapped to can be triggered.

Other common elements that define the behaviour of a runnable of process are *LabelAccess* and *Ticks*. These accesses are generalized by the abstract class *ComputationItem*. While the former allows one to specify a data access, the latter describes the number of instructions that must be performed during execution. To gain variability in the model and, thus, to represent the actual software behaviour in a more precise manner, a varying number of instructions per execution can be defined via different distribution functions.

Also, like RunnableCall, the classes for the access of semaphores or labels refer the accessed objects. In addition to that, these classes contain an enumeration which describes the kind of access, e.g., read/write.

Processes and process prototypes

In addition to the *Process* element, which generalizes interrupt service routines (ISRs) and tasks, the AMALTHEA model contains an element *ProcessPrototype*. This prototype can be used to define raw data of a process. It can be used to specify access to labels (read, write) or other *Runnables* as possible with the normal *Process*, but not the order of the access. A prototype is then processed by algorithms. The algorithms are creating the processes, are filling, verifying, or modifying the data based on their different checks. The results of this processing are at the end processes, which are corresponding to the data of the prototypes.

These processes are representing the current state and can be further processed, for example, to generate code or further simulation. With the process prototypes available in the model, it is possible to define the structure of the software in an early development phase. The

implementation at that moment is open and not yet completed, but the general idea can be verified. Another motivation is the distribution of software from a single-core system to a multicore system. Therefore, the activity graph can be analysed and computed to get the right order and parallelisation of the elements and dependencies.

5.5.3 Operating system

This part of the AMALTHEA model describes the provided functionality of an operating system. It mainly provides a way to specify how access to certain system resources is given. Therefore, the concepts of scheduling, buffering, and semaphores are supported, which are described in detail in the following.

Scheduler

A scheduler controls the execution of processes on a core of a processor. The controlled core is referred by the scheduler object. The processes are mapped to the scheduler. Each scheduler is assigned an algorithm according to which the resources are managed:

- Fixed Priority
- OSEK
- FixedPriorityPreemptive
- FixedPriorityPreemptiveWithBudgetEnforcement
- DeadlineMonotonic
- RateMonotonic
- Dynamic Priority
- EarliestDeadlineFirst
- …

OS Overheads

To describe a more detailed and realistic behaviour, the overhead that is produced by an operating system can be defined. These overheads can be assigned to an operating system definition. Each overhead information is defined as a set of instructions that must be executed when the corresponding OS function is used. The instructions can be either a constant set or a deviation of instructions.

Semaphore

With this object, a semaphore can be described which limits the access of several processes to one resource at the same time.

5.5.4 Stimuli

A stimulus is responsible to activate processes and there are different types of it. The stimulus of type SingleStimulus activates the process only one time. The InterProcessStimulus defines an activation based on an explicit inter process activation. The periodic stimulus has an offset and a recurrence time. The first activation occurs at the offset time. Every following activation occurs after recurrence time. With a jitter, it is possible to define a deviation of time for every activation. For this, a periodic stimulus can define a time deviation which can be a gauss distributed, a Weibull distributed or a uniform distributed deviation. This deviation is always

positive. So, it increases the time of an activation. In addition to that it is possible to shift the activation to left or right on the timeline. In this way it is possible to increase or decrease the activation time again.

First, there is a periodic stimulus with a fix offset and recurrence time. The first activation occurs after offset, every following activation occurs after recurrence. In the second diagram is the same stimulus with a gaussian stimulation deviation. The activation varies now, but the minimum time between the activation is always the time of recurrence. In the third line, a negative shift is added to stimulation deviation. Now it is possible that the time distance between activations is less than the time of recurrence.

Besides a deviation, so called scenarios can be defined. A scenario is a specific manifestation of clocked stimuli that describes the progress of time for one or more variable rate stimulus in relation to global time. If two equal stimuli have a different time base, the time of task activation can be different. There are different kind of clock functions the clock sinus function, the clock triangle function, and the clock multiplier list. The clock multiplier list is a list of timestampmultiplier value pairs. Is a specified timestamp arrived, the clock changes to the corresponding multiplier value.

5.5.5 Mapping

Mapping contains the principle to distribute the defined software structure to available hardware elements. The software defines different requirements or constraints to consider by doing this type of distribution. These constraints are mainly the following:

- Core constraints: Software elements can define the needed type of a core to run on. This can include for example a required floating point unit. The core constraint targets mainly the execution time.
- Memory constraint: Defines the preferred or needed type of a memory, for example a flash type.

The core and memory constraints are in general coming from the developer or architect of the software elements, as they are having the knowledge about the contained implementation and the corresponding requirements for the environment. Algorithms, which are responsible to process the distribution, must be able to handle the defined constraints.

5.5.6 Timing

In this section the meta-model for formalizing information related to timing is described. That way, a solution for handling timing information during all steps of the design process of embedded real-time systems is provided. Therefore, the existing domain-specific language Timing Augmented Description Language (TADL), developed in the TIMing MOdel (TIMMO) project, respectively TADL2 the results from the successor project TIMing MOdel - TOols, algorithms, languages, methodology, and USE cases (TIMMO-2-USE) was adapted. Not only does it harmonize with EAST-ADL, an Architecture Description Language (ADL) for automotive embedded systems, which was maintained by the European FP7 project MAENAD, but also the standardized automotive software architecture AUTOSAR. 'The primary purpose of TADL

is the enrichment of design models that are created by typical modelling languages like EASTADL2 and AUTOSAR with data that specify the required and/or existing timing behaviour of the parts of the systems described'. This makes the analysis of a given system in terms of timing and dynamic behaviour possible. Structural modelling is performed on higher abstraction levels like the vehicle, analysis, or design level as defined within EAST-ADL. For modelling on the implementation level, the AUTOSAR meta-model is used. Timing constraints are then defined separately from the structural modelling.

For the definition of timing constraints structural elements are referred to using the concept of events and event chains. Since different sets of events are available for EAST-ADL and the AUTOSAR model, events in TADL are specially tailored to refer to specific elements in the structural model. Thus, the syntax of TADL is compliant to the released AUTOSAR as well as to EAST-ADL meta-model. In the following all necessary elements for modelling timing constraints are presented:

Event

For analysing the internal behaviour of a real-time system, it is necessary to receive notifications about what happened at which instance of time. Therefore, information about the current state of a running system and its entities as well as knowledge of the moments in which a specific state occurs or changes is needed. Properties of real time entities or objects that stay valid during a finite duration are called state attributes. Thus, each of these attributes has at least one possible state. But due to the dynamic nature of embedded systems these entities are constantly subject to modifications. Therefore, the occurrence of such a change of state is denoted as event. Based on the appearance of events, it is possible to reconstruct the temporal behaviour and causal interactions within a system. AMALTHEA knows quite a few different Events for indicating specific incidents in the system:

- ProcessEvent
- ProcessChainEvent
- StimulusEvent
- RunnableEvent
- LabelEvent
- ChannelEvent
- SemaphoreEvent • ComponentEvent

Event Chain

According the definition of an event chain, a pair of events that must be causally related. This means a single event chain consists of at least the following two events:

- stimulus: indicates the beginning of the chain
- response: terminates the event chain

Event chains that are more complex and do not only consist of a stimulus and a response are strictly ordered by the position of the events in the chain defined. That means, sequences with the same set of events but a different order of event occurrences are not equal. However, only the stimulus and response event can be defined directly in an event chain. Chains that take more than those two events into account must be divided into segments, whereas each segment represents a single event chain again. The same procedure applies for defining

alternative paths from the stimulus to the response. An event chain can, therefore, contain one or many strands, which represent the alternative paths. Like segments, these strands are event chains themselves again.

Timing Constraints

So far it is just possible to describe causal relationships within real time systems. To analyse the temporal quality of these relations however, it is necessary to express some kind of restriction based on timing expressions. Therefore, AMALTHEA defines the model of timing constraints. An event constraint describes the basic characteristics of the way an event or event chain occurs over time. Thus, every timing constraint has at least one timing expression, which then is applied to the referenced event or event chain. AMALTHEA defines the following set of basic constraints, which will be presented in detail in the following:

- Delay Constraint: This constraint describes how a source and a target event are placed relative to each other. The constraint describes a 'window', which is spanned through lower and upper values.
- Repetition Constraint: A repetition constraint describes the distribution of a single event during runtime. It allows one to constraint that a second succeeding instance of the start event of a task must occur within a specific time window after the initial start event. Furthermore, a jitter for each event can be stated.
- Synchronization Constraint: Like the delay constraint, the synchronization constraint is a restriction based on the distance between the occurrences of events. Unlike the delay constraint, where the distance between a source and a target event is constrained, the synchronization constraint considers a group of events and limits the distance of the events within this group.
- Order Constraint: The order constraint defines a sequence of events. The restriction is based, as the name implies, on the correct order of the occurrences according to the specified position. The order constraint initially implied a constraint between two timing expressions.

There is, however, a whole set of other constraints available, which are all basically derived from one of the above constraint forms.

5.6 Data Model parts for non-functional validation and verification

This section describes relevant existing data models (only the relevant parts) for V&V methods to be considered in WP5. As mentioned in the introduction, this is mainly focused on timing and not on safety or security.

We have evaluated various data models upon their possibilities to model timing-relevant data and requirements specifications. We therefore defined criteria to be assessed for each data model and collected the results in a table. Table shows the resulting overview. In the following we will provide some details about the criteria and the data models.

Criteria	**Candidate specification/modelling languages**				
	AMALTHEA APP4MC	**AUTOSAR**	**EAST-ADL2**	**Temporal Logic**	**TIMMO2 (TADL2)**

1.	**Suitability to generate code (WP2.4 and 2.5)**	L (Service needs WP2.5)	L (Service needs WP2.5)	VL (structural code)	Observer Generation only	VL (same as EASTADL2)
2.	**Meeting our use case requirements**	H	VH	L	--	M
3.	**Expressiveness**	M	M	M	VH	M
4.	**Fitness for testing (testability)**	VH	H	M	M (via observers)	H

Table 5-1: Timing information in data models

We chose some criteria to evaluate the suitability of considered data models. For each criterium we have five different values: very low (VL), low (L), medium (M), high (H), very high (VH). These values are somewhat generic and serve as a rough assessment for overview. Although course, the relative nature of the assessment enables a systematic comparison. The criteria are:

1. Suitability to generate code: assess if the data model provides enough detail/information so that we can generate code based on the information
2. Meeting our use case requirements: asses if the data model can be used to validate/verify all use case specific requirements (this assessment is currently dynamic, since the use cases still define their requirements)
3. Expressiveness: assesses the variability in expressing requirements and future requirements which can currently not be anticipated
4. Fitness for testing: assesses if it is possible to derive test cases to test the system either based on the model or based on an implementation (e.g. via simulation, tracing, etc.) The following paragraphs will give detailed descriptions on the assessment of the criteria for each considered data model.

5.6.1.1.1 AMALTHEA/APP4MC

Timing simulation is one tool for model-based timing verification. AMALTHEA/APP4MC is built for timing simulation and can be used to express more simulation specific details compared to AUTOSAR (e. g. probability distributions for execution times). Since it is primarily a meta-model to represent the timing relevant data of a system in a model, it is not very well suited to generate code from it. The behaviour, for example, is not represented at all. AMALTHEA can be used to verify and validate a system via simulation, so at least parts of the use cases are covered (the avionic use case demands guarantees for some verification activities – these cannot be given by a simulation or testing approach). You can use AMALTHEA to analyse the timing properties of the system to provide guarantees, however, there might be data missing that is relevant but cannot be represented easily with AMALTHEA. A direct consequence of AMALTHEA being mainly purpose-built for timing and performance simulation is that it is limited in expressiveness. For instance, one can only specify timing events that are visible at the known artefacts of the meta-model. Custom requirements thus have to be explicitly supported by a meta-model extension (which is possible since the APP4MC implementation of AMALTHEA is open sourced under the EPL2). Lastly, the testability of an AMALTHEA model is very high (with respect to its supported requirements). Essentially, there are two kinds of

tests possible: simulation trace evaluation and target trace evaluation. These encapsulate about the vast majority of all timing related tests.

5.6.1.1.2 AUTOSAR

Depending on where in the development process one uses AUTOSAR, it can support in code generation. However, only the structural code (i.e. code skeletons) is generated. Similar to AMALTHEA, there is no support for behaviour code generation. We assessed AUTOSAR to be highly suited for our use case since it is a requirement for the automotive demonstrator and there are many tools natively supporting the AUTOSAR exchange format. The envisioned tool integration for the functional architecture modelling tool and the timing simulation tool will use AUTOSAR as a widely adopted standard. Considering the expressiveness, AUTOSAR is on par with AMALTHEA. Even though AUTOSAR supports more artefacts for timing event specification it offers roughly the same kinds of requirements/constraints on timing events and event chains. Since AUTOSAR can be extended by the consortium, it is generally possible to adapt the specification, although this involves an extensive specification process. Again, like AMALTHEA, AUTOSAR is suited for testing. We evaluated the testability with "high" because AMALTHEA offers a more unambiguous interpretation of its requirements, and AUTOSAR does not define specific metrics as testing targets.

5.6.1.1.3 EAST-ADL2

EAST-ADL2 is a rather high-level architecture description language. With this in mind, it is questionable if it is even suited to generate structural code at all. That is why we assessed the code generation suitability with very low. There is no requirement from the WP1 use cases to use EAST-ADL2, and there are only a few tools supporting it. So, EAST-ADL2 does not meet our use cases very well. Judging the expressiveness of EAST-ADL2, it is the same as AUTOSAR and AMALTHEA and offers roughly the same timing events, event chains, and requirements. There are some specialities (like the BurstConstraint or OutputSynchronizationConstraint), but in practice there are mostly not considered. The fitness for testing is medium mainly because of the high abstraction level (thus, many assumptions would have to be made to support testability).

5.6.1.1.4 Temporal Logic

Temporal Logic is not a data model per se. It is a semantically well-defined language to formulate properties that a system must possess. The system is thereby usually given as a set of parallel composed state machines (e.g. for timing this can be timed automata and their derivates). For this reason, temporal logic can be used to generate observer code quite well, however, the other half of it (the entities that shall be observed) must support the appropriate interfaces to provide observable information. It does support giving guarantees for verification of timing requirements. In practice, temporal logic is relatively difficult to formulate on higher abstraction levels. Therefore, it is only used where guarantees are required, and we do not expect that the prerequisites and method meet the use cases in XANDAR. Since temporal logic is a language, its expressiveness is very high (which, in turn, is part of the reason why it is relatively difficult to formulate it) – so depending on the expressiveness of the state machines it can be quite powerful for verification. One can use temporal logic for testing, by only considering parts of the state space of the modelled system. Additionally, one can use the

generated observers to observe if the system under test behaves correctly. So, we assessed the testability with medium.

5.6.1.1.5 TADL2

TADL2 was defined in the TIMMO2USE research project. TADL1 served as a blueprint for the Timing Extensions in AUTOSAR, whereas TADL2 was almost directly incorporated into EASTADL2. TADL2 itself is only the meta-model to specify timing properties and requirements. Therefore, it is difficult to assess whether it can be used for code generation, similar to EASTADL2. It is a potential fit for our use cases, however, depends greatly on the meta-model it is attached to. If, for example, the XANDAR consortium decides to use a custom meta-model to specify the system under design, it may be straightforward to attach TADL2 to it to specify timing properties and requirements. In that sense, it meets our use cases. However, this can not be assessed at this point in time and implications for tool support and interfaces are unclear. Like the other meta-models described above, we assessed TADL2 to be medium when it comes to expressiveness – they are all very similar. The suitability for testing (depending on the meta-model it is attached to) can also be high.

In summary, the evaluated data models are (at least semantically) mostly supported by our timing validation methods (existing and planned ones). One notable exception is temporal logic: Currently, we do not plan to extend our timing validation and verification analyses to also cover model checking via timing annotated state machines and temporal logic. The remaining (model based) data models, however, are good candidates to source input data from for our planned granularity levels for timing analyses (see Subsection 4.2.7 in D5.1).

6 Existing tool interfaces

6.1 Vector PREEvision

PREEvision provides different kinds of imports and exports designed for exchanging product data in the automotive industry (specifically between OEMs and suppliers) [2]. While some of the import and export formats have become quasi-standards (such as AUTOSAR and FIBEX) others are proprietary and serve as internal communication media (such as EPDM).

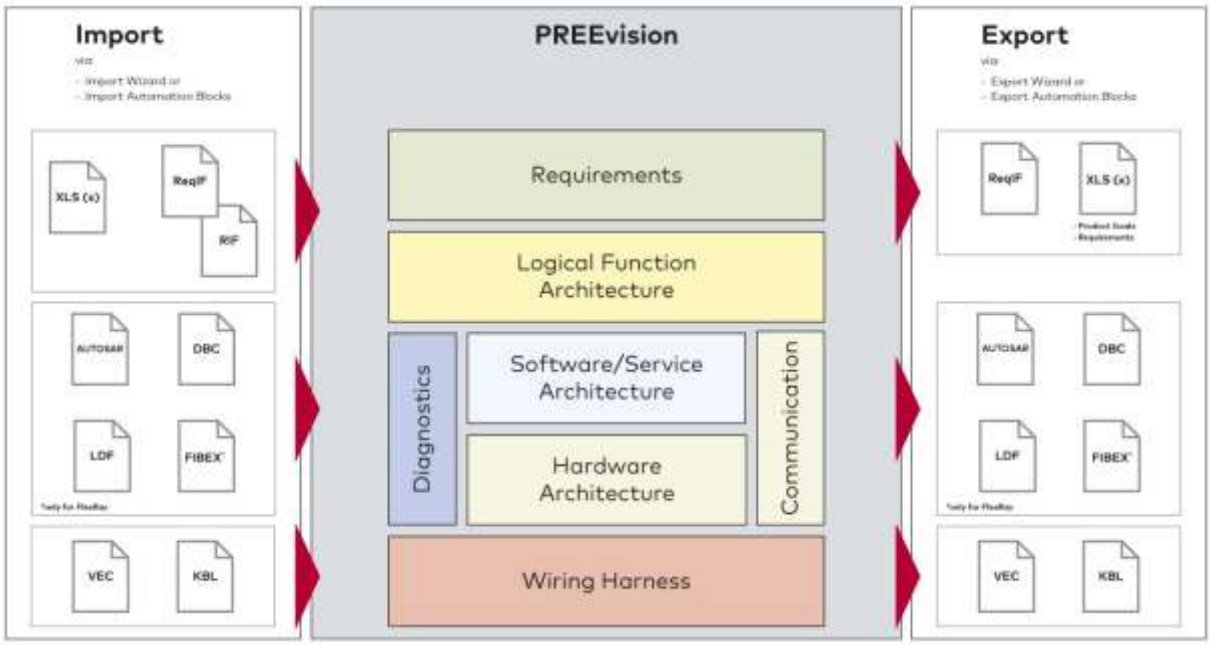

Figure 6-1: PREEvision: Import and export overview

The following table lists standards and its versions that are supported by the PREEvision version 10.0 import and export:

Standard	Availability and version
AUTOSAR System Description	• AUTOSAR import (4.0.3, 4.1.1, 4.2.1, 4.2.2, 4.3.0, 4.3.1, 4.4.0) • AUTOSAR export (4.0.3, 4.1.1, 4.2.1, 4.2.2, 4.3.0, 4.3.1, 4.4.0) Since the import is downward compatible, older versions that are not listed may be accepted. AUTOSAR files of newer AUTOSAR versions are rejected during import.
AUTOSAR ECU extract	• AUTOSAR export (4.0.3, 4.1.1, 4.2.1, 4.2.2, 4.3.0, 4.3.1, 4.4.0)
AUTOSAR Adaptive System Description	• AUTOSAR import (19-03) • AUTOSAR export (19-03)
DBC	• DBC import • DBC export
Excel (for requirements)	• Requirements excel import and export The Excel import is for requirements and has a generic approach. The import options are saved in an import configuration file that can be selected in the

	import dialog. Generally, you can define an own Excel format and describe the mapping from your Excel format to PREEvision artefacts in the configuration file. A default Excel format with mappings is provided with PREEvision.
FIBEX	• FIBEX import (3.0.0, 3.0.1, 3.1.0) • FIBEX export (3.1.0)
KBL	• KBL import (2.3, 2.4) • KBL export (2.3, 2.4)
VEC	• VEC import (1.1.3) • VEC export (1.1.3)
LDF	• LDF import (2.0, 2.2) • LDF export (2.0, 2.2) • J2602 import and export
ReqIF (Requirements Interchange Format)	• ReqIF import (1.0.1, 1.1, 1.2) • ReqIF export (1.2)
RIF (Requirements Interchange Format)	• RIF import (1.1.a, 1.2)

Table 6-1: Standards and versions supported by PREEvision import and export

6.2 Vector TA Tool Suite

6.2.1 AMALTHEA

As described in Subsection 5.5, the AMALTHEA exchange format contains several parts of a timing model to describe the dynamic behaviour of a system. The most important ones include the Software, Hardware, Stimuli, Operating System, Mapping, Constraints, and Events model artefacts.

The software model artefacts are supported straight forward. However, there are some parts that are not supported in the import/export of the software model artefacts: Process Chains, Process Prototypes, Activations, Sections, and Channels.

From the hardware model artefacts, the tool suite supports the following: System Description and Access Path. The generic nature of the type descriptions (networks, cores, ECUs, systems, memories, and microcontrollers) is not preserved, however, the properties of the types are imported into the corresponding instances of the tool suite. For the export way, only the natively supported type properties of the tool suite are exported as types. In that respect, some information may get lost when doing a roundtrip (model import followed by an export). Since the tool suite supports the representation and simulation of vendor specific processors (e.g. the Infineon AURIX TC29x series) while AMALTHEA does not (at least not natively), some hardware model artefacts from the tool suite are not exported to AMALTHEA.

For operating system specification, the tool suite supports OS overheads, semaphores, the generic operating system, and the vendor operating system. Here, the same issues as in the hardware model emerged for vendor specific operating systems (e.g. Elektrobit AutoCore). However, in AMALTHEA it is possible to specify vendor operating systems which can then be correctly imported into the tool suite for simulation.

The stimuli model parts of AMALTHEA are supported straight forward as well as the mapping model artefacts. During the import into the tool suite, however, the runnable mappings from the mapping model are excluded. Software Components, as specified by AUTOSAR (see

Subsection 5.4), are partly supported by AMALTHEA, so the im- and export is also supported in the tool suite. In the tool suite, however, they are represented in a different container called "Architecture". Among the components this package is also used to represent safety groups which are called Physical Sections in AMALTHEA. In that respect, AMALTHEA and the tool suite's internal meta-model diverge.

For tool-assisted optimization, model changes, and evaluation activities, there are constraints in AMALTHEA, such as latency constraints, runnable sequencing constraints, or data age constraints. Given a correct configuration of a system in the aforementioned model parts, these constraints can be evaluated by specialized tools (like the tool suite). They can also be used to change this configuration, upon which the constraints are re-evaluated to decide whether the change produced a valid configuration.

To simplify the use of the APP4MC Eclipse application to produce tool-suite-ready models (AMALTHEA models that can correctly be loaded by the tool suite), there are special validation rules. If these special tool suite validations do not report any issue, the import will work.

6.2.2 AUTOSAR System Description import

A system description typically contains the software design of the whole functionality, the communication network, and the SWC-to-ECU mapping. It may also contain additional constraints. The system hardware description is given as several ECUs which may include ECU partitions. The following table shows which AUTOSAR artefacts are imported and interpreted.

AUTOSAR artefact	**TA Tool Suite artefact**	**Mapping Description**
ApplicationPartition	-	There is no direct representation, however, application partitions are used to determine the SWC-To-Core mapping. So, they are necessary for the task to core allocation as well. Note: Along with the mapping, the Application Partition is mandatory for a simulation.
SW-ComponentType	Software-Component	TATS software components are a direct translation, however, they only *refer* to runnables, whereas AUTOSAR SWCs *contain* their runnables
P-PortPrototype/ R-PortPrototype	Port	The property "direction" determines whether it is a provided or a receiving port. The required/provided interface can be either a Sender-Receiver- or a Client-Server-Interface, see below for interface import.
SWC-InternalBehavior	-	This artefact is not imported directly, all runnables and events contained in an internal behaviour are imported. Note: The internal behaviour (and thus its runnables) will only be considered in case the behaviour is owned by an AtomicSW-Component-Type.
Runnable-Entity	Runnable	Directly imported, all Data-Read-Accesses and Data-WriteAccesses are imported as corresponding signal accesses. The order of the signal accesses is the same as in the AUTOSAR file. If there are TimingEvents specified for the Runnable-Entity, they will be imported as Stimuli.

Timing-Event	Periodic	**Basic Task**: If a Runnable-Entity has exactly one TimingEvent, then a corresponding Periodic stimulus will be created (unless a stimulus with that exact period and offset exists already). If no Task, which is stimulated by the Periodic

		stimulus, exists, a new one will be created and will be mapped to that stimulus. This basic task will always have an MTA (maximum concurrent task activations) of 10, a priority of 10, and will be a foreground task. The runnable will then be added to the call sequence of the task. **Extended Task**: If a Runnable-Entity has more than one Timing-Event, then the corresponding Periodic stimuli will be created (if they do not exist yet). Along with these periodic stimuli OS-Events will be created which will be set at each stimulus occurrence. Then, an extended foreground task with MTA 1 and period 1 will be created. For this task, a single stimulus will be created to stimulate the task once, after that it will run in an endless loop. Within this endless loop there are points at which the execution will wait for one of the OS-Events to be set. Thus, the corresponding runnable will be stimulated by multiple different periodic activations.
Init-Event	Single	Creates a single stimulus with optional offset.
DataReceivedEvent/Data-SendCompletedEvent/Data-Write-CompletedEvent	OSEvent	In case the data is part of a Sender-Receiver-Interface (in which case it will be mapped to an InterfaceSignal), an OSEvent is created which will be set if the InterfaceSignal is received.
Activation-Reasons	-	Will be ignored.
Data-ReadAccess/DataWrite-Access	Signal-Access	Data-Read and Data-Write accesses will be imported as sender/receiver accesses. The import will firstly create the corresponding Sender-Receiver-Interface (if it does not exist yet) and will add the accessed Signal to that interface if it is not yet part of it. Secondly, a SignalAccess with the corresponding access kind (read or write) will be created for the owning runnable. Note: Only one-to-many communication is supported (one sender, many receivers).
Composition-SW-ComponentType	Software-Composition	Translates directly to SoftwareComposition with its required and provided ports.

SW-Connector	Port-Connector	Our internal connection representation does not

v1.0		flat-extract or derive a flat composition hierarchy, 63

v1.0		we do not need to consider multiple instantiations

SWC-	-	Not considered directly, only used to import

v1.0	1.0	implementatio n	resource consumptions (runnable runtimes).

v1.0	ResourceConsumption/	Weibull-	Measured and simulated execution times are

v1.0	Execution-Instructions	imported as Weibull distributed execution times.

v1.0		The nominal time (which must be between

v1.0		minimum and maximum) specification is imported 74

v1.0	Uniform-	Analysed execution times are imported as	77

v1.0		corresponding min and max values. Min has to be

v1.0	Constant-	The rough estimate is imported as Constant	82

System	-	The system is not imported as an artefact;

v1.0	Root-SW-	Software-	The root SW composition prototype in the system 87

v1.0	Composition-	Composition	is used as a starting point to import the whole 88

v1.0	Prototype		software component hierarchy. Types that are not	89

v1.0	System-	Task-Scheduler,	A task scheduler will be created for each core (an	91

Mapping	Task-Mapping	ECU- Partition that has at least one application partition mapped to it) if at least one software component is mapped to the core (at least one SW-Component-Prototype is mapped to the application partition). For each task (derived from activation pattern of runnables in software components) determine the application partition it is mapped to (use SWC-To-Application-Partition-Mapping to determine which runnables in software components are mapped to which application partitions). Then this N tasks to 1 ECU-Partition can be directly interpreted as task-to-core scheduler mapping.
ECU-Instance	Execution-Unit, Operating-System	Each ECU instance is imported as an execution unit. For each execution unit at least one operating system is created. If there are multiple processors (determined by admin data) an operating system for each processor will be created.
ECU-Partition	-	Will be ignored.
SenderReceiver-Interface	SenderReceiver-Interface	During the traversal of the signal read and write accesses the sender receiver interfaces will be created as mentioned above. This is a direct translation.
SWC-Timing, VFB-Timing	-	These container elements are considered during the import but are not imported as an element. They refer to a specific software component or VFB data access upon which timing extension artefacts are defined.
TD-EventSWC-InternalBehavior	TimingEvent	Events e.g. "runnable entity activated" will be imported as the TimingEvent equivalent. Depending on the context in which this timing event is defined (VFB or SWC-Timing) the events will be created accordingly.
TD-Event-Variable-Data-Prototype	TimingEvent	This is translated as a signal access (read or write) event that happens in a software component on a certain port.
TimingDescriptionEvent-Chain	TimingEventChain	Event chains with their stimulus/response event references as well as their segments are directly imported as the corresponding TimingEventChains.
Execution-Time-Constraint	Runnable-Requirement	Two different runnable requirements will be created, one for the minimum and one for the maximum allowed execution time.
Latency-Timing-Constraint	EventChain-LatencyConstraint	This is a direct translation for required latency of event chains.
Periodic-Event-Triggering	Periodic-Constraint	This is again a direct translation for the required periodic activation behaviour of runnables.
Age-Constraint	DataAge-Constraint	In case the age constraint refers to a port on a software component a data age constraint will be created that refers to the signal access event.
Offset-TimingConstraint	Delay-Constraint	Creates a delay constraint with the minimum and maximum time that is allowed to pass between two events.

Table 6-2: AUTOSAR artefacts imported by the Vector TA Tool Suite

6.2.3 Trace formats

In order to consolidate simulated traces with measured traces the tool suite supports trace imports. The comparison of traces helps in validating the model and thus improves future simulation results. Since tracing is not a major topic in XANDAR, we shortly list the supported and most relevant trace formats in the following.

6.2.3.1 Best Trace Format (BTF)

This trace format is a text-based representation of a trace that is natively supported by the tool suite. The specification is open and free and is available on the APP4MC documentation website [23]. Currently, many other tracing-related companies and their tools support them (e.g. INCHRON, Luxoft, avelabs, iSYSTEM, etc.). Though, the specification does not provide a machine-readable specification format (i.e. a grammar), it is well documented how to implement the BTF specification.

6.2.3.2 Runtime Measurement Module

Runtime Measurement (RTM) is a Vector specific BSW module that allows the user to determine runtimes and CPU load of BSW modules and user code sections. Measurement is controlled- and evaluated in CANoe by RTM's frontend or a self-written RTM application. Data exchange between CANoe and the ECU is done by the XCP protocol (e.g. using CAN or FlexRay network communication). Gathered results of a measurement by CANoe can be exported into a CSV file for a tool suite import.

6.2.3.3 What to do with the imported traces

During the trace import of the various trace formats, the tool suite derives some model entities from the information contained in the traces. That means, depending on the amount of data contained in the trace format, we can reconstruct a model from the trace. With the original model and simulation trace, it is then possible to relate trace and model events from the reconstructed model to the original ones. It is also possible to derive runtime information from the traces – this, however, only produces runtimes as "seen" in the trace at hand.

6.3 Ptolemy II

Figure 6-2 shows an excerpt from the XANDAR development process in which all steps related to the verification of functional behaviour are highlighted. As visualized in the figure, this activity is based on the Ptolemy II simulator performing a combined execution of the system and the environment model. From a tool interface perspective, it is therefore necessary to specify how information required to perform such a simulation is supplied to Ptolemy II.

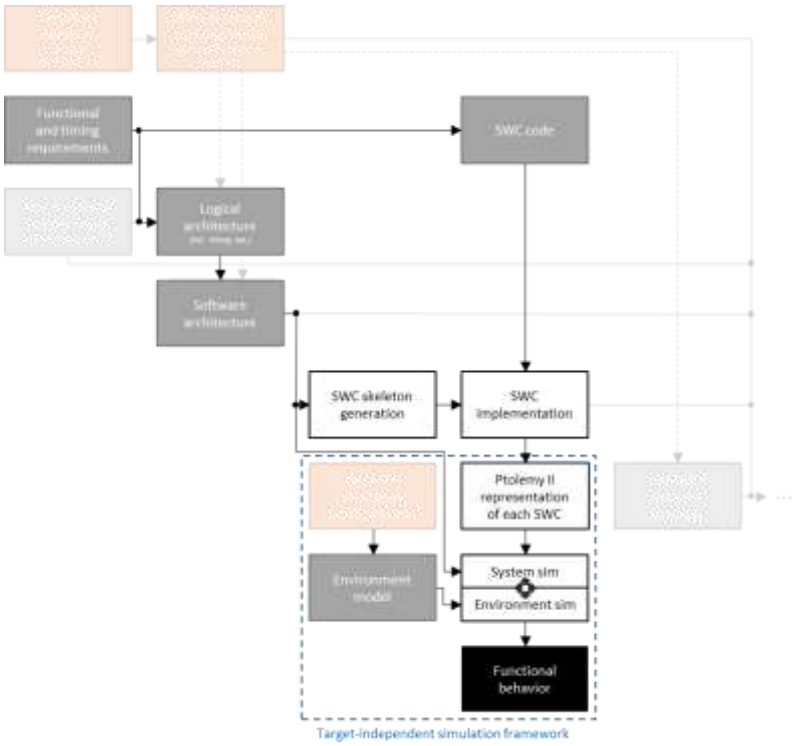

Figure 6-2: Process steps related to the verification of functional behaviour

Specifically, the simulation depends on information from the three following sources:

1. **Software architecture:** This artefact models the network of SWCs that the system is made up of. This includes information on all input and output ports as well as the manner in which these ports are interconnected at system level.
2. **SWC implementations:** An executable representation of the behaviour that every SWC will exhibit. For a more detailed description, refer to D3.1 of this project.
3. **Environment model:** A set of executable model fragments, each representing a particular environment in which the system model can be integrated to perform a MiL or SiL simulation.

Ptolemy II models are stored in an XML format called *Modeling Markup Language* (MoML). This language is available as an XML document type definition (DTD) and further described in [24]. It is particularly capably of representing hierarchical models and allows users to define model portions as a so-called class. Classes can then be instantiated from, i.e., hierarchically integrated into, another model.

The environment model, which is considered to be a part of the target-independent simulation framework, is expected to be supplied by the user of the XANDAR toolchain. To derive such a model, a suitable MoML file need to be created, for example through Vergil, the graphical modelling frontend provided as part of Ptolemy II.

The system model needs to be made available to users in such a way that they are able to integrate it into the environment model. Therefore, the software architecture will be transformed into a suitable MoML class. An important property of this class is that it that will be created in such a way that it contains one dedicated Ptolemy II actor per SWC that is part of the software architecture. Each of these actors will consist of auto-generated Java code that makes use of

the interface described in section 12.4 of [1] to incorporate the behaviour of the relevant SWC implementation into the system model.

7 Conclusions

A detailed description of the XANDAR process has been given above. Modelling languages / standards and their capabilities have been discussed regarding the process. The modelling languages presented have been identified based on the toolchain and the requirements originating from it. The tool introduction provides a starting point for readers to gain insights into the practical application of the XANDAR toolchain.

Based on the modelling requirements listed above and the current capabilities of the tools PREEvision and Vector TA Tool Suite, the interfaces of these two tools will be extended. This extension shall enable the timing verification use case on system level, to support the timingby-design approach in XANDAR. To that end, timing modelling capabilities will be adjusted in PREEvision and the Vector TA Tool Suite will import and work with these new artefacts. These XANDAR extensions will be detailed in a later deliverable.

The feasibility of the additional requirements is being investigated in the XANDAR project. This investigation is ongoing, including thorough analysis of the project use cases from the automotive and aviation domains This may yield a more fine-grained specification of the requirements, to be detailed in later documentation. According to the results of this analysis, necessary extensions of other tools and their interfaces will be specified and implemented.

8 References

[1] Claudius Ptolemaeus (Editor), System Design, Modeling, and Simulation using Ptolemy II, Ptolemy.org, 2014.

[2] Vector Informatik GmbH, *PREEvision version 10.0 User Manual,* 2021.

[3] E. A. Lee, "Heterogeneous actor modeling," in *2011 Proceedings of the Ninth ACM International Conference on Embedded Software (EMSOFT)*, Taipei, Taiwan, 2011.

[4] E. A. Lee, "CPS foundations," in *Proceedings of the 47th Design Automation Conference*, Anaheim, California, 2010.

[5] EAST-EEA Project Partners, "Deliverable D3.6: EAST-EEA - Embedded Electronic Architecture," EAST-EEA Consortium, Konstanz, Germany, 2004.

[6] P. Cuenot, P. Frey, R. Johansson, H. Lönn, Y. Papadopoulos, M.-O. Reiser, A. Sandberg, D. Servat, R. Tavakoli Kolagari, M. Törngren and M. Weber, "The EAST-ADL Architecture Description Language for Automotive Embedded Software," in *Model-Based Engineering of Embedded Real-Time Systems: International Dagstuhl Workshop, Dagstuhl Castle, Germany, November 4-9, 2007. Revised Selected Papers*, H. Giese, G. Karsai, E. Lee, B. Rumpe and B. Schätz, Eds., Springer Berlin Heidelberg, 2010, p. 297–307.

[7] Object Management Group, "OMG Unified Modeling Language (UML), Version 2.5.1," December 2017. [Online]. Available: https://www.omg.org/spec/UML/2.5.1/PDF. [Accessed 18 March 2021].

[8] International Organization for Standardization (ISO), *ISO 26262: Road vehicles – Functional safety,* 2011.

[9] M. Peraldi-Frati, A. Goknil, J. DeAntoni and J. Nordlander, "A Timing Model for Specifying Multi Clock Automotive Systems: The Timing Augmented Description Language V2," in *2012 IEEE 17th International Conference on Engineering of Complex Computer Systems*, 2012.

[10] Object Management Group, "OMG Systems Modeling Language (SysML), Version 1.6," November 2019. [Online]. Available: https://www.omg.org/spec/SysML/1.6/PDF. [Accessed 18 March 2021].

[11] Object Management Group, "OMG Requirements Interchange Format (ReqIF), Version 1.2," July 2016. [Online]. Available: https://www.omg.org/spec/ReqIF/1.2/PDF. [Accessed 18 March 2021].

[12] AUTOSAR initiative, "AUTOSAR Website - Standards," October 2018. [Online]. Available: https://www.autosar.org/standards/classic-platform. [Accessed 18 March 2021].

[13] AUTOSAR, *Specification of ECU Resource Template R20-11,* München: AUTOSAR GbR, 2020.

[14] AUTOSAR, *Software Component Template R20-11,* München: AUTOSAR GbR, 2020.

[15] M. Krause, O. Bringmann, A. Hergenhan, G. Tabanoglu and W. Rosentiel, "Timing Simulation of Interconnected AUTOSAR software-components," in *Design, Automation and Test in Europe Conference*, San Jose, CA, USA, 2007.

[16] AUTOSAR, *Glossary R20-11,* München: AUTOSAR GbR, 2020.

[17] AUTOSAR, *Basic Software Module R20-11,* München: AUTOSAR GbR, 2020.

[18] AUTOSAR, *System Template R20-11,* München: AUTOSAR GbR, 2020.

[19] AUTOSAR, *Specification of ECU Configuration R20-11,* München: AUTOSAR GbR, 2020.

[20] AUTOSAR, *Specification of Timing Extension R20-11,* München: AUTOSAR GbR, 2020.

[21] AUTOSAR, *Specification of Timing Extension for Adaptive Platform R20-11,* München: AUTOSAR GbR, 2020.

[22] AUTOSAR, *Specification of Abstract Platform R20-11,* Munich: AUTOSAR GbR, 2020.

[23] Eclipse Foundation/Commiters, "Documentation of the APP4MC project," 29 01 2016. [Online]. Available: https://www.eclipse.org/app4mc/documentation. [Accessed 12 May 2021].

[24] E. A. Lee and S. Neuendorffer, *MoML — A Modeling Markup Language in XML — Version 0.4,* 2000.

[25] R. Hebig, "Methodology and Templates in AUTOSAR," February 2009. [Online]. Available: http://www.hpi.uni-potsdam.de/fileadmin/hpi/FG_Giese/Ausarbeitungen_AUTOSAR0809/Methodology_hebig.pdf.