Document Title	Timing Analysis and Design
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	645

Document Status	published
Part of AUTOSAR Standard	Foundation
Part of Standard Release	R23-11

Document Change History

Date	Release	Changed by	Description
2023-11-23	R23-11	AUTOSAR Release Management	• Added System Level Logical Execution Time • Reworked functional level use-cases in chapter 4 • Updated TIMEX to ARTI mapping in appendix B • Updates on use-cases and improvements
2022-11-24	R22-11	AUTOSAR Release Management	• New introduction of timing on functional level in 4 • Added description of Timing Reference Platform on functional level(TRP) in appendix A • Reworked end-to-end, network and ECU use-cases. • Minor updates and improvements
2021-11-25	R21-11	AUTOSAR Release Management	• Added “Timing Requirements and Abstraction Levels” in section 2.1 • Extended description of Timing Reference Platform (TRP) in appendix A • Added TIMEX to ARTI mapping in appendix B • Minor updates and improvements

△

2020-11-30	R20-11	AUTOSAR Release Management	–Migration of document to standard "Foundation"– • Added description of Timing Reference Platform (TRP) in appendix A. • Minor updates and improvements • Editorial changes
2019-11-28	R19-11	AUTOSAR Release Management	• Added section 5.9 on introduction of service oriented communication • Minor updates and improvements • Editorial changes • Changed Document Status from Final to published
2018-09-31	4.4.0	AUTOSAR Release Management	• Extended section 1.4 to show interaction of AUTOSAR CP and AP concepts • Reworked chapter structure for better readability • Added description of AUTOSAR CP task states and extended timing parameter table in section 9.1.1.1 and section 9.1.1.2 • Added chapter 10 including timing tasks and elements
2017-12-08	4.3.1	AUTOSAR Release Management	• Editorial changes

▽

△

2016-11-30	4.3.0	AUTOSAR Release Management	• Section 1.10 added roles and their benefits from reading this document • Section 4.2 introduced function-level Use-cases • Some ECU UCs are consolidated in chapter 7 • New figure for overview of E2E Use-cases is improved (figures 5.1) • Improved timing tasks in section 10.1 • References to methods and properties are consolidated in chapter 9
2015-07-31	4.2.2	AUTOSAR Release Management	• Section 10.1: introduced basic timing tasks like “Collect Timing Requirement” or “Create Timing Model”. Adapted introduction of chapter 9 accordingly. • Clarified relation of the timing properties described in section 9.4 to AUTOSAR TIMEX. • improved glossary and index • New figures for improved overview of use-cases (figures 7.2 and 6.2)
2014-10-31	4.2.1	AUTOSAR Release Management	Editorial changes only: improvements, corrections and additions. • New chapter End-to-End Timing for Distributed Functions; • Chapter Properties and Methods for Timing Analysis: additional information and restructuring; • Added further use-cases; • Added examples, see figures 1.2, 7.1 and 6.1; • Added index at the end of the document;
2014-03-31	4.1.3	AUTOSAR Release Management	• Initial version

Disclaimer

Contents

References

[1] Methodology for Classic Platform
AUTOSAR_CP_TR_Methodology

[2] Specification of Timing Extensions for Classic Platform
AUTOSAR_CP_TPS_TimingExtensions

[3] Explanation of Adaptive Platform Design
AUTOSAR_AP_EXP_PlatformDesign

[4] Software Process Engineering Meta-Model Specification
http://www.omg.org/spec/SPEM/2.0/

[5] Embedded Systems Development, from Functional Models to Implementations

[6] System-level Logical Execution Time:Augmenting the Logical Execution Time Paradigm for Distributed Real-time Automotive Software

[7] The evolution of real-time programming

[8] Giotto:a time-triggered language for embedded programming

[9] EAST-ADL - Model Domain Specification
http://www.east-adl.info/Specification.html

[10] Tool Support for the Analysis of TADL2 Timing Constraints using TimeSquare
http://hal.inria.fr/docs/00/85/06/73/PDF/paper.pdf

[11] Unified Modeling Language:Superstructure, Version 2.0, OMG Available Specification, ptc/05-07-04
http://www.omg.org/cgi-bin/apps/doc?formal/05-07-04

[12] System Modeling Language (SysML)
http://www.omg.org/spec/SysML/1.3/

[13] UML Profile for Modelling and Analysis of Real-Time and Embedded systems (MARTE)
http://www.omg.org/spec/MARTE/1.1/

[14] Architecture Analysis and Design Language (AADL) AS-5506A
http://standards.sae.org/as5506a/

[15] TIMMO-2-USE

[16] Specification of Operating System
AUTOSAR_CP_SWS_OS

[17] Methodology for Adaptive Platform
AUTOSAR_AP_TR_Methodology

[18] Scheduling algorithms for multiprogramming in a hard real-time environment
http://cn.el.yuntech.edu.tw/course/95/real_time_os/present paper/Scheduling Al-

gorithms for Multiprogramming in a Hard-.pdf

[19] Controller Area Network (CAN) Schedulability Analysis:Refuted, Revisited and Revised
http://dl.acm.org/citation.cfm?id=1227696

[20] Pushing the limits of CAN-Scheduling frames with offsets provides a major performance
http://www.loria.fr/ nnavet/publi/erts2008_offsets.pdf

[21] Probabilistic response time bound for CAN messages with arbitrary deadlines
http://ieeexplore.ieee.org/xpl/abstractAuthors.jsp?arnumber=6176662

[22] The Esterel synchronous programming language:design, semantics, implementation

1 Introduction

This document represents recommended methods and practices for timing analysis and design within the AUTOSAR development process. It is intended for different kinds of readers:

- system, development and test engineers with no or little knowledge of timing analysis
- engineers with general knowledge of timing analysis who want to enhance their understanding of AUTOSAR methodology
- further stakeholders (listed under 1.10)

1.1 Objective

During the development of AUTOSAR based systems, a common technical approach for timing analysis is needed to fulfill the AUTOSAR main requirement RS_Main_00340. This document describes all major steps of timing analysis needed from the definition and validation of functional timing requirements to the verification of timing requirements on component and system level. Figure 1.1 illustrates the different aspects for timing analysis. Basis for the described methods are AUTOSAR Methodology [1] and AUTOSAR timing extensions [2].

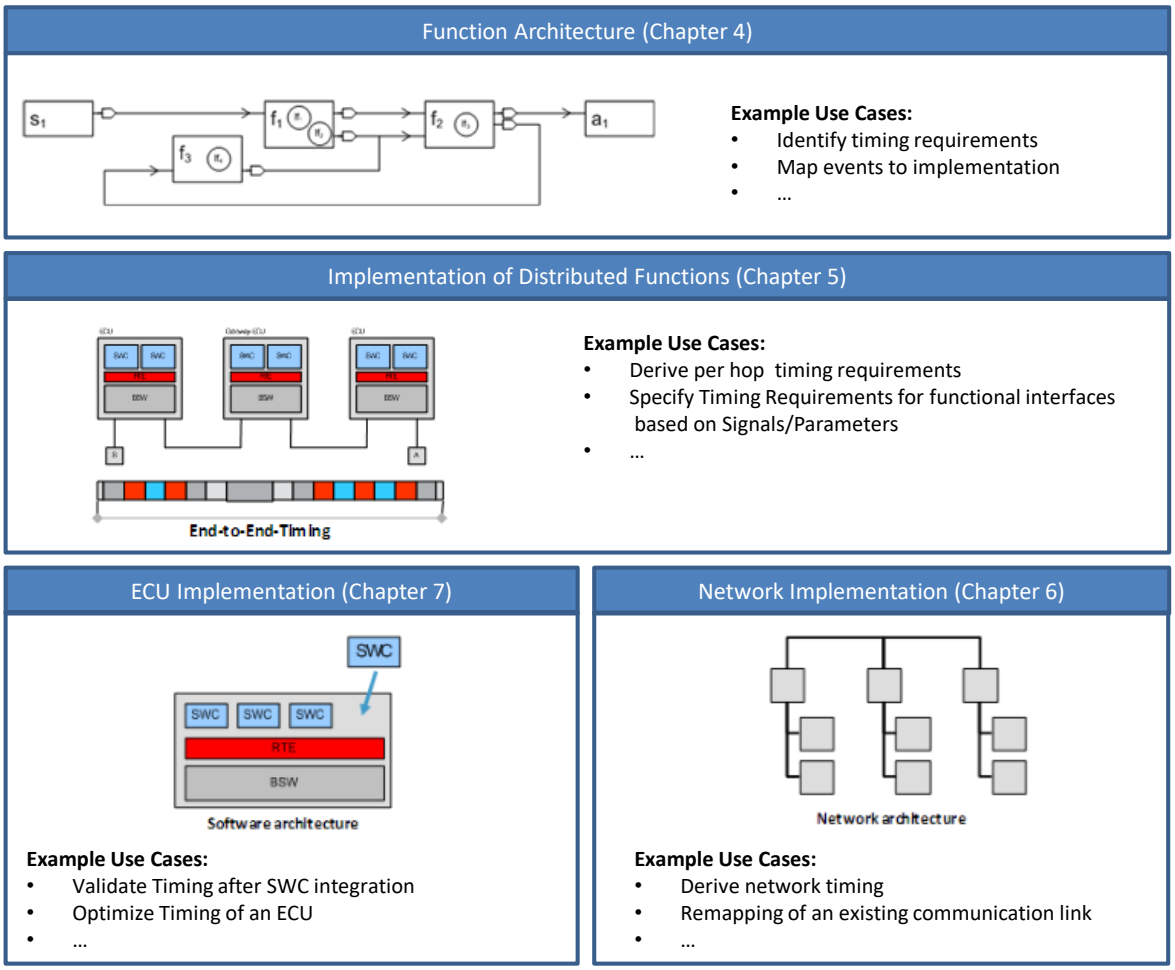

Figure 1.1: Overview of aspects for timing analysis

1.2 Overview

The AUTOSAR timing analysis methodology is divided in following parts:

- Decomposition of timing requirements
- Timing analysis on function level
- End-to-end timing analysis for distributed functions
- Timing analysis on the network level
- Timing analysis on the ECU level
- Timing properties and methods for timing analysis

For each part, a proposed methodology is presented based using a number of typical real world use-cases. A complete overview of all use-cases is given in section 1.9 on page 23.

1.3 Motivation

The increasing number of functions, complexity in E/E Architectures and the resulting requirements on ECUs and communication networks imply increasing requirements on the development process. A central part of the development process is the design of robust and extendible ECUs and network architectures.

In the development of ECUs complexity is introduced through the integration of multiple SW-Cs (constituting various functions) executed in schedulable tasks. The design and verification of the task schedules becomes difficult due to their dependencies on shared resources such as processing cores and memory.

On the network level heterogeneous network types such as CAN, LIN, FlexRay, MOST and Ethernet are used. This makes it hard to ensure robustness, especially when routing between protocols over a gateway takes place. The design of an efficient and robust network architecture and configuration is increasingly difficult. This creates the need for a systematic approach.

These aspects must be addressed in the E/E development process together with additional requirements regarding quality, testability, ability to perform diagnostic services and so on. The overall goal is to achieve sufficient reliability and performance at optimum cost under the requirement of scalability over several vehicle classes. In order to enable integration of additional functions over the life-cycle of a vehicle, the extensibility of an E/E architecture is also very important.

To make optimal technical decisions during the development of E/E architectures and their components it is necessary to have suitable criteria to decide how to implement a function.

One of the most important criteria in the development of current E/E architectures is timing. Many functions are time critical due to their safety requirements. Other functions have certain timing requirements in order to guarantee a high quality (customer) function. These functions often have certain latency and jitter constrains. For distributed functions these constraints consist of several segments of which ECU and network are the two main categories. In order to specify and analyze these timing requirements functional timing chains are important. These are described in more detail in Chapter 3.

1.4 Example

The active steering shown in the figure 1.2 demonstrates an end-to-end timing constraint with a real-world AUTOSAR Classic Platform (CP) example. The system consists of sensors, ECUs, buses and an actuator. With the vehicle dynamics model of the car and the active steering function the functional developer defined a maximum reaction time for the outlined chain: 30ms. This becomes a top level end-to-end timing requirement for the system.

This timing requirement then gets decomposed, i.e. it gets sliced into smaller portions T1...T5, one portion for each component of the system. Obviously, ECUs and buses handle many different features with their own timing requirements, all competing for network and computation resources. On an ECU with tasks/interrupts and their runnables, the top level timing requirements are broken down into more fine grained timing requirements and the competition for resources is continued on a lower level.

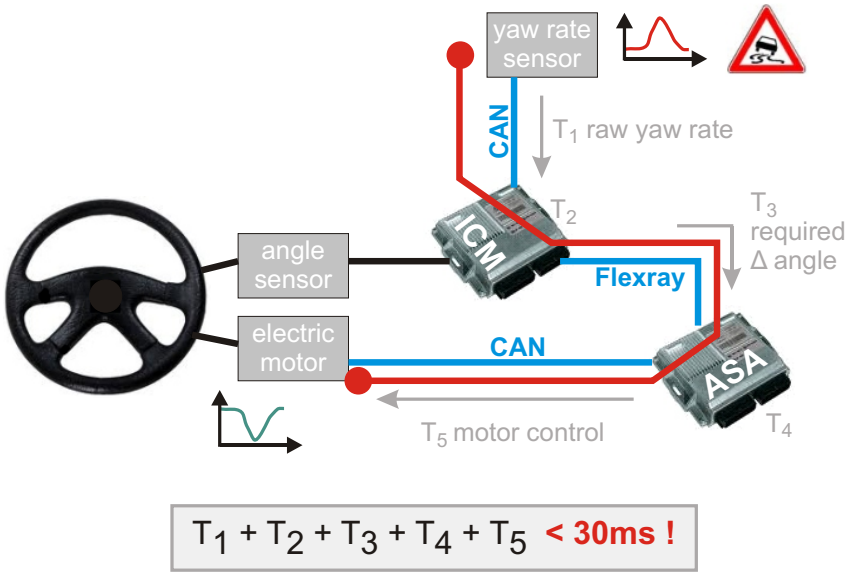

Figure 1.2: Set-up and end-to-end-timing requirement (red line) from an active steering project.

In this example, the embedded software is developed independently from the later allocation on concrete ECUs, i.e. ICM and ASA in Figure 1.2. First the functionalities that should be covered by the system are defined and subsequently transfered into a software architecture. A possible AUTOSAR software architecture representing the active steering example can be found in Figure 1.3.

The example consist of seven AUTOSAR software components communicating via sender receiver ports. First, the system determines data about the vehicle and environment such as vehicle speed, steering angle, and environmental disturbance (such as yaw rate). This information is provided to the motion arbiter that rates the situation and deduces further activities of the vehicle actuators accordingly. Depending on the input data a deceleration command, an acceleration request, and/or an updated steering direction can be sent to further components.

The executed commands directly influence the wheel speeds and the steering angle. Thereby, the driving program (actuating variable) and the environmental disturbance, e.g. the yaw rate, is controlled. Altogether, software, hardware and environment form a feedback control system. AUTOSAR Classic Platform caters specifically to hard real-time systems like this one.

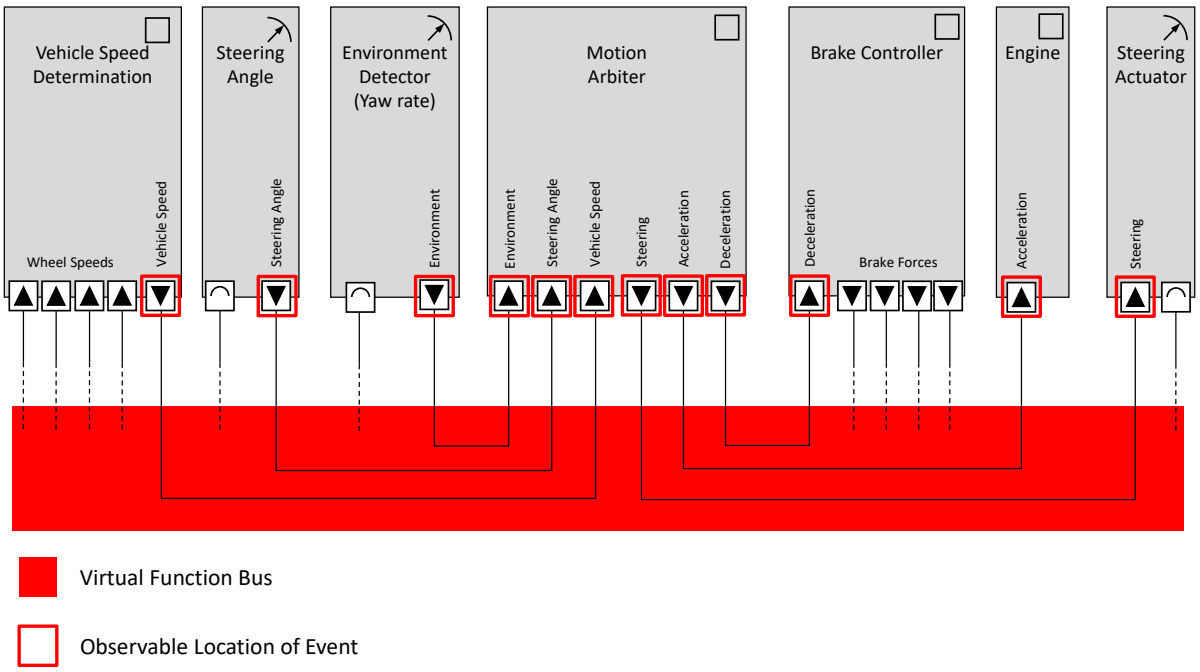

Figure 1.3: Software architecture of the above introduced active steering project.

When considering modern assisted-driving functions, the example above can be extended by adding a collision avoidance system which uses computer vision to recognize obstacles and directs the steering to circumvent them. Recognizing objects from camera images and planning appropriate avoidance trajectories are computationally demanding requirements which are very hard to implement using only AUTOSAR CP. This kind of application is specifically targeted by AUTOSAR Adaptive Platform (AP). This extension adds a second top level end-to-end timing requirement for the system. The collision avoidance system needs to recognize an obstacle and a clear path around it, plan an appropriate trajectory and issue required angle commands to the ASA quickly enough to avoid collision.

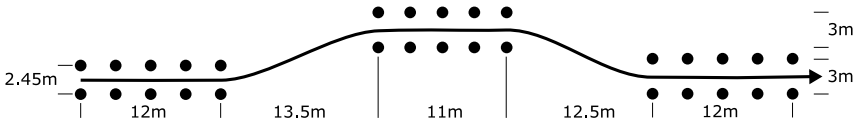

Figure 1.4: ISO 3888-2 "elk test" schematic overview

Based on the ISO 3888-2 evasive maneuver (Figure 1.4) this results in a TA1-TA2-TA3-T4-T5 decomposition with the TAx components taking place in the AP domain (See Figure 1.5). At 14m/s (roughly 50kph) TA1...TA3 will have a budget of 860ms (12m at 50kph) for object detection, trajectory planning and communication of the first required angle adjustment to the ASA. The requirement of the duration of T4+T5 is 10ms, based on maximum safe steering gradient and vehicle dynamics, in order to fulfill the lane change requirement of ISO 3888-2 within 13.5m of longitudinal movement. Note that both the CP and AP requirements share the same T4 and T5 due to both control loops sharing the same actuator path.

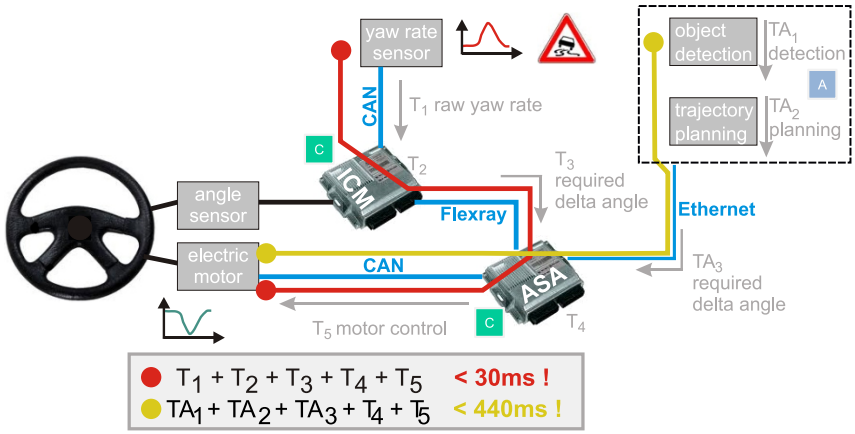

Figure 1.5: Active steering project augmented by camera based obstacle avoidance (AUTOSAR Adaptive Platform).

The possible AUTOSAR software architecture representing the extended active steering example can be found in Figure 1.6. A more thorough discussion of the integration of AUTOSAR CP and AP ECUs can be found in Explanation of Adaptive Platform Design [3].

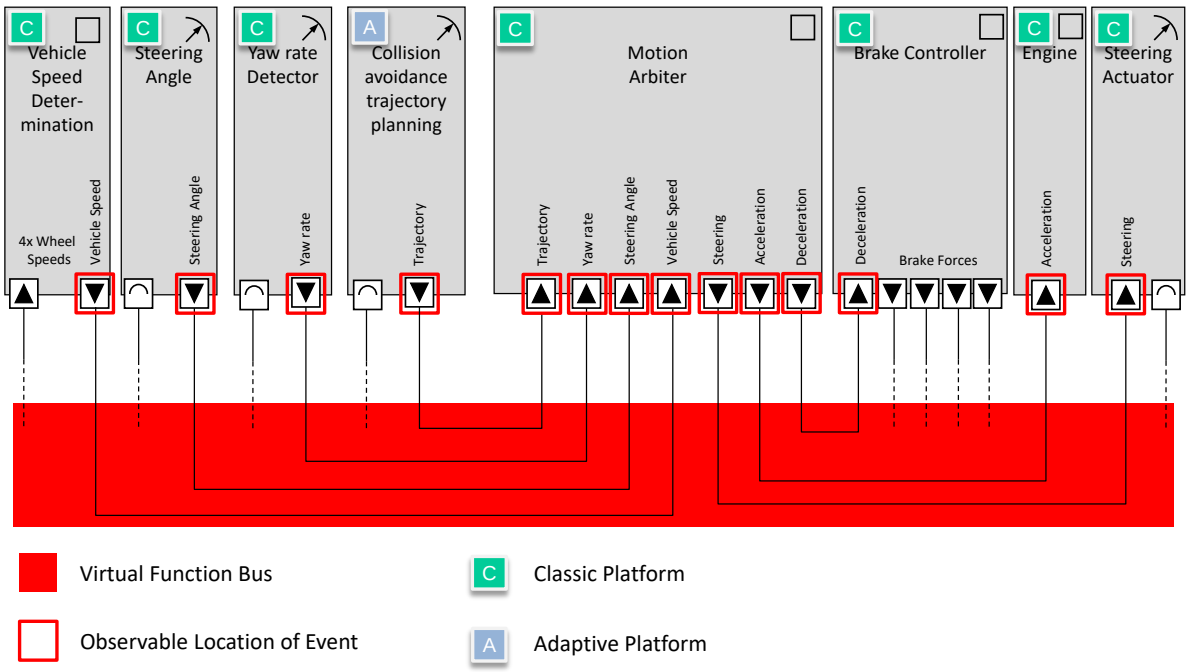

Figure 1.6: Software architecture of the above introduced active steering project.

1.5 Scope

This document describes how to implement timing analysis during the development of E/E systems. Similar to [1], this does not include a complete process description but rather a set of practical methods to define timing requirements and how to ensure that

these requirements are met. As stated in [1], the methodology is designed to cover the needs of various AUTOSAR stakeholders:

- Organizations: Methodology is modeled in a modular format to allow organizations to tailor it and combine the methodology within their own internal processes, while identifying points where they interact with other organizations.
- Engineers: Methodology is scoped to allow engineers of various roles quickly find AUTOSAR information that is relevant to their specific needs.
- Tool Vendors: Methodology provides a common language to share among all AUTOSAR members and a common expectation of what capabilities tools should support.

The following topics are addressed:

- Definition of appropriate timing analysis methods including related timing properties for all stages of an AUTOSAR development process without disclosure of company confidential information.
- Definition of requirements for timing analysis methods enabling implementation of appropriate tools.
- Documentation of relevant experience in the area of timing analysis (Network and ECU/software) with relevant use-cases.
- Structuring of timing tasks, timing properties and related methods with regard to use-cases.
- Timing as an enabler for efficient cooperation on a functional level between OEM and tier1.

Delimitation:

- Contents of this document is complementary, and not overlapping, to the contents of the AUTOSAR timing extensions [2]
- Definition of meta models to document timing attributes (e.g. AUTOSAR TIMEX)
- Definition of timing behavior for specific SW-Cs or functions in AUTOSAR.

1.6 Acronyms and Abbreviations

Abbreviation	*Meaning*
ASA	Active Steering Actuator
AUTOSAR	AUTomotive Open System ARchitecture
BSW	Basic Software
CAN	Controller Area Network
COM	Communication module
CPU	Central Processing Unit
DES	Discrete Event Simulation

E2E	End to end
ECU	Electrical Control Unit
ICM	Integrated Chassis Management
ID	Identifier
I/O	Input/Output
LIN	Local Interconnect Network
NW	Network
PIL	Processor-In-The-Loop
PDU	Protocol Data Unit
RE	Runnable Entities
RTE	Runtime Environment
SW-C	Software Component
SPEM	Software Process Engineering Meta-Model
TD	Timing Description
TIMEX	AUTOSAR Timing Extensions [2]
UC	Use-Case
UML	Unified Modeling Language
WCET	Worst case execution time
WCRT	Worst case response time
VFB	Virtual Functional Bus

Table 1.1: Acronyms and Abbreviations

1.7 Glossary of Terms

Term	*Synonym*	*Definition*
Event-triggered `Frame`	Sporadic `Frame`	A frame that is sent on an event triggered by the application independent from a communication schedule. The event-triggered sending is limited by a debounce time which specifies the shortest allowed temporal distance between two occurrences.
Accuracy		The accuracy is the closeness to the true value. For the worst case of a timing property it describes the maximum overestimation.
Cause-Effect Chain		A cause-effect chain represents the data-flow among communicating components, by relating read events of a consumer component to the corresponding write events of a producer component.
Execution Time		The execution time is the total time that the function needs to be assigned the resource in order to complete.
Event Chain		An event chain describes a causal order for a set of functionally dependent timing events. (See TimingDescriptionEventChain in [2])
`Frame`	Message	A frame is a data package sent over a communication medium. This element describes the structure of data (OSI layer 2) sent on a channel. For example, a frame on CAN and FlexRay. A commonly used synonym is “message”.
Hyperperiod		The hyperperiod is the least common multiple of all periods in a system.
Information Packages		Smallest transmittable information unit on a resource (e.g. frame).
Interconnect LET		ref. Section 8.1.
Interrupt Load		The load of the CPU for servicing interrupts.

LET Task		ref. Section 8.1.
Load	Utilization	The load is the total share of time that a resource is used. Please note that within the context of this document the terms load and utilization are used synonymously. The term load as in the number of users waiting for a resource to become available, is not considered in this document.
Logging		Logging is the activity of providing arbitrary, not necessarily correlated, informational data by software. Logging collects information to understand the behavior of one or multiple programs running on a real system. In contrast to Tracing, the focus is on collecting information explicitly added by a software developer on source code level. Based on the requested Log Level, logging may have an timing and/or load impact on the system, which has to be considered during further analysis. Examples: Error logging, Printf output, ara::log
Period		The time period between two activation events of the same frame(network) or task(ECU).
Response Time	Latency	Response time is the time between the occurrence of an event until it is processed. E.g. The time between the transmission request of a message until its reception or the time between activation of a function and its completion.
Schedulable Entity		A schedulable entity is defining an execution that can occupy time on a CPU or on a network resource. The order of execution is decided by scheduling algorithms. Schedulable entities are for example tasks, processes and frames.
Stuff Bit		In CAN frames, a bit of opposite polarity is inserted after five consecutive bits of the same polarity.
System Parameter		A quantity influencing the timing behavior of the system.
Timing Task		A number of steps to accomplish a specific goal (see 9 “Description of Timing Tasks”).
Timing Constraint		A timing constraint may have two different interpretation alternatives. On the one hand, it may define a restriction for the timing behavior of the system (e.g. minimum (maximum) latency bound for a certain event sequence). In this case, a timing constraint is a requirement which the system must fulfill. On the other hand, a timing constraint may define a guarantee for the timing behavior of the system. In this case, the system developer guarantees that the system has a certain behavior with respect to timing (e.g. a timing event is guaranteed to occur periodically with a certain maximum variation). Compare AUTOSAR Timing Extension [2]
Timing Method	Technique	Defines an ordered number of steps to derive particular timing related work products (e.g. timing property, timing model)
Timing Model		A timing model collects all relevant timing information in one single place, typically tool-based. The model can be used to describe the timing behavior or it can be used to generate timing related configuration files.

Timing Property		A timing property defines the state or value of a timing relevant aspect within the system (e.g. the execution time bounds for a `RunnableEntity` or the priority of a task). Thus, a property does not represent a constraint for the system, but a somehow gathered (e.g. measured, estimated or determined) or defined attribute of the system.
Tracing		Tracing is the activity of recording run-time information over a certain period of time by observing a real system. Tracing collects events of selected types over time and stores the information persistently in a so called "trace buffer". For proper timing measurement, the events may be stored together with a time stamp. Depending on the tracing method, the trace buffer may be on-board or off-board. Depending on the trace method, tracing may or may not have a timing impact on the system. If it has, the impact has to be considered when doing further analysis. The recording may be done by software solutions (e.g. code instrumentation), hardware assisted solutions (e.g. CPU instruction flow tracing, Ethernet sniffers) or a combination of them. The trace buffer may be analyzed and visualized offline, providing information about the internal behaviour of the system. Examples: ARTI, VFB Tracing, L&T
Use-case	Scenario	Typical problem, broken down into tasks
Worst case		The term “worst case” denotes an upper bound on any value a certain property can take during run-time. This is usually different from and may never be smaller than the maximum value observed in the actual system. Typically worst-case values are derived using static analyses based on models of the system.
Work Product		See SPEM [4].

Table 1.2: Glossary of Terms

1.8 Limitations

One of the key features of the AUTOSAR Adaptive Platform is adaptability. Applications can be started and stopped on-the-fly. Existing applications updated or even new applications installed over-the-air. This results in the possibility of operation conditions changing rapidly and unpredictably. It is no longer possible to predict and analyze the timing for all possible operating conditions.

Currently the scope of this document is limited to analysis and design of a well-known system until its delivered. Future extensions will cover possibilities for timing analysis of systems with a high levels of uncertainty introduces by the adaptability of the AUTOSAR Adaptive Platform.

1.9 Use Cases

In order to show the proposed usage of timing analysis methodology a number of real-world use-cases are included in the document.

The use-cases are divided into categories using the same structure as the chapters:

- Timing analysis on the function level (chapter 4)
- End-to-end timing analysis for distributed functions (interface between ECU and network level) (chapter 5)
- Timing analysis on the network level (chapter 6)
- Timing analysis on the ECU level (chapter 7)

Section	Use-case	Page
4.2	Overview of Function-level Use Cases	56
5.2	Overview of End-to-End Use Cases	64
6.2	Overview of Network Use-cases	81
7.3	Overview of ECU Use Cases	104

Table 1.3: List of all use-cases in this document

1.10 Methodology Roles

This section introduces roles that can benefit from knowledge about the methods presented in this document and will be used in the Timing Analysis Methodology.

Role	**ECU Integrator**		
Package	AUTOSAR Root::M2::Methodology::Methodology Library::Common Elements::Roles		
Brief Description	Integrates the complete software on an ECU.		
Description	Integrates the complete software on an ECU, which includes generating necessary code and completing the configuration of all software components and basic software modules.		
Benefit	Receives information about how to define standardized timing requirements (related to the function) and how to verify them.		
Relation Type	***Related Element***	***Mul.***	***Note***
Performs	TBC	1	n.A.

Table 1.4: ECU Integrator

Role	**E/E Architect**		
Package	Not in the AUTOSAR methodology yet. A part of AUTOSAR System Engineer Role.		
Brief Description	Defines E/E topology.		
Description	Defines E/E topology.		
Benefit	Receives information about how to evaluate the timing quality of the E/E architecture under timing requirements (resources and timing budgets, high level).		
Relation Type	***Related Element***	***Mul.***	***Note***

Performs	TBC	1	n.A.

Table 1.5: E/E Architect

Role	**Function Architect**		
Package	Not in the AUTOSAR methodology yet.		
Brief Description	Defines (high level) timing requirements for the function.		
Description	Defines (high level) timing requirements for the function.		
Benefit	Receives information about how to define standarized timing requirements (related to the function) and how to verify them.		
Relation Type	***Related Element***	***Mul.***	***Note***
Performs	TBC	1	n.A.

Table 1.6: Function Architect

Role	**Function Engineer**		
Package	Not in the AUTOSAR methodology yet. Must be adapted from AUTOSAR System Engineer Role.		
Brief Description	Defines and decomposes timing requirements.		
Description	Defines timing requirements at system level, decomposition of E2E timing requirements into local timing requirements and function can be implemented, resp. content of the transferred data, makes partition.		
Benefit	Receives information on how to define, refine and decompose timing requirement related to the function, E2E etc. under condition of a correct implementation and test, can reason about the implications of integrating a subsystem into a vehicle.		
Relation Type	***Related Element***	***Mul.***	***Note***
Performs	TBC	1	n.A.

Table 1.7: Function Engineer

Role	**Network Data Engineer**		
Package	Not in the AUTOSAR methodology yet.		
Brief Description	Defines communication matrix, Frames, PDUs, Triggerings, Network Management, Routing Matrix, content -> data		
Description	Defines communication matrix, Frames, PDUs, Triggerings, Network Management, Routing Matrix, content -> data		
Benefit	Receives information about the mapping of the function architecture to the communication matrix on networks under timing and resource aspects (Use cases chapter 4).		
Relation Type	***Related Element***	***Mul.***	***Note***
Performs	TBC	1	n.A.

Table 1.8: Network Data Engineer

Role	**Software Architect**
Package	Not in the AUTOSAR methodology yet.
Brief Description	Refines timing requirements to SW implementation level, decomposition of timing requirements down to the implementation
Description	Refines timing requirements to SW implementation level, decomposition of timing requirements down to the implementation

Benefit	Learns how to consider timing and use time budgeting on SW-Cs when mapping runnables to tasks.		
Relation Type	***Related Element***	***Mul.***	***Note***
Performs	TBC	1	n.A.

Table 1.9: Software Architect

Role	**Software Component Developer**		
Package	AUTOSAR Root::M2::Methodology::Methodology Library::Common Elements::Roles		
Brief Description	Developer of the software component code.		
Description	Develops the SW-C internal behavior, which means the code executing the function of a SW-C. He respects the interfaces to other SW-Cs and knows about functional and timing requirements for the function he engineers.		
Benefit	Gets in contact what the requirements given for developing the SW-C internal behavior are used for. With this knowledge he can develop the code more verification-friendly and identify requirement conflicts. Using his system knowledge he can enhance the requirement set and consult other roles.		
Relation Type	***Related Element***	***Mul.***	***Note***
Performs	TBC	1	n.A.

Table 1.10: Software Component Developer

Role	**Test Engineer**		
Package	Not in the AUTOSAR methodology yet.		
Brief Description	Performs measurements and timing related tests.		
Description	Performs measurements and timing related tests.		
Benefit	Receives information how to carry out timing analysis and verification on the system, Information about methods and properties.		
Relation Type	***Related Element***	***Mul.***	***Note***
Performs	TBC	1	n.A.

Table 1.11: Test Engineer

Role	**Timing Engineer**		
Package	Not in the AUTOSAR methodology yet.		
Brief Description	Performes timing analysis and verification.		
Description	Creates timing model, performs timing analysis, proves the timing results against the timing constraints, resp. tools, models.		
Benefit	Receives information on how to model a system and how to carry out timing analysis and verification on the model (using different methods).		
Relation Type	***Related Element***	***Mul.***	***Note***
Performs	TBC	1	n.A.

Table 1.12: Timing Engineer

1.11 Document Structure and Chapter Overview

This section contains an overview of the document and the chapter contents. Figure 1.1 on page 14 illustrates the different aspects for timing analysis and indicates the chapters in which these will be addressed. In order to show relevance in real world systems, each aspect is described based on one or more typical use-cases, which are linked to Methodology Roles in Chapter 1.10. These use-cases are split into smaller timing tasks. For each of these tasks the necessary timing properties and the corresponding timing methods are presented. These are used to validate the timing and performance constraints typical for the corresponding use-case.

Chapter 1 “Introduction” contains the objective, motivation, scope of the document abbreviations and glossary of terms. Additionally, a list of the use-cases is contained in section 1.9.

Chapter 2 “Basic Concepts of Timing” gives a general overview of timing analyses and introduces the relevant elements and concepts.

Chapter 3 “Timing Requirements on Design Levels” contains a short introduction about the challenge of breaking down functional timing requirements from an abstract user’s view to the implementation view of AUTOSAR timing extensions. The problem definition, different approaches and concepts for methodological solutions are introduced.

Chapter 4 “Timing on Functional Level” describes timing-related use-cases for system function analysis and design on functional level. Some use-cases are covering the high-level timing in an early stage of the development while others are dealing with the transition from the functional level to the implementation level. This chapter is intended for E/E Architects and Function Architects.

Chapter 5 “End-to-End Timing for Distributed Functions” introduces the techniques and methodology to reason about the end-to-end timing of distributed functions. They can consist of a locally executed function that uses data from distributed sources (e.g. sensor data) or the computation itself can be distributed. Typical constraints are latency, period and data age. This chapter is intended for E/E Architects, Function Architects and Function Engineers.

Chapter 6 “Timing for Networks” contains use-cases for applying timing analysis at network level, covering scenarios such as extension of an existing network, design of a new network or redesign/reconfiguration of existing network architectures. This chapter is addressed mainly to Network Data Engineers and ECU Integrators.

Chapter 7 “Timing for SW-Integration on ECU Level” contains use-cases for applying timing analysis at ECU level. The chapter covers several use-cases with different levels of abstraction covering the complete development workflow of an ECU ranging from creating a timing model of the entire ECU up to timing optimization. For every use-case the corresponding methods and timing properties are linked. This chapter is addressed mainly to Software Architects and ECU Integrators.

Chapter 9 “Properties and Methods for Timing Analysis” covers the timing tasks, timing properties and the methods derived from the use-cases. Every single method is presented in detail including its classification, description, relation to use-cases, requirements, timing properties, inputs, boundary conditions and its implementation. Some of the methods deliver timing properties as output which can be evaluated by means of timing constraints to check the fulfillment of the timing requirement. Every single timing property is characterized by its classification, description, relation to use-cases, requirements, timing methods, format, (valid) range and implementation. The methods can be grouped in three main groups: simulation, analytical calculation and measurement; whereas the properties can be separated in two main groups: latency-like and bandwidth-like. An overview of the relation between the single methods and the single timing properties respectively is given, but also the interaction between the two is outlined.

In chapter 10 “Artifacts for Timing Analysis” the artifacts (e.g. timing tasks, work products) from the use-cases are collected. Additionally common elements for a timing model and timing-related work products are described.

2 Basic Concepts of Timing

2.1 Timing Requirements and Abstraction Levels

Timing properties described in the previous section have to be taken into account all along the specification and design process. In the specification phase, these timing properties are expressed as timing requirements on the system functions and decomposed during the system specification and design phases. In order to achieve this decomposition, it is better to follow some methodological principles. The following chapters of this document will present in more details these principles through the description of use cases. This section gives the definitions of the main timing abstraction levels considered in this document. It also gives some preliminary rules for applying timing requirements decomposition throughout these abstraction levels.

2.1.1 Timing Abstraction Levels

In this document, we will consider the following main timing abstraction levels corresponding to different levels of abstraction of platforms:

- **Functional Level** This abstraction level is out of AUTOSAR modeling scope but is of primary importance to capture timing requirements from the early specification phases. It consists in an abstract architecture of the system functions.
- **Abstract Platform Level** This abstraction level is in the AUTOSAR modeling scope. It consists of a description of an architecture of abstract components. These components are platform agnostic and can be mapped to any concrete platform (AUTOSAR or non AUTOSAR platforms).
- **Concrete Platform Level** This abstraction level corresponds, in the scope of AUTOSAR, to Classic Platform or Adaptive Platform, and can also correspond to a non AUTOSAR concrete platform (e.g. COVESA).

Timing requirements decomposition has to be coherent between these levels of abstraction. A minimal set of guidelines shall be followed to ensure this coherence. Among these guidelines, it is important to distinguish concepts of decomposition and transformation.

2.1.2 Chaining Decompositions and Transformations

In this document, we will consider two kinds of transitions: transformation and decomposition. These are used in the following context and definitions.

- **Decomposition** A decomposition consists in splitting a component or a timing requirement (more generally an artifact) at a given level of abstraction (either

at functional-level, or abstract platform level or at a concrete platform level). A decomposition is a result of a design or organizational decision.

- **Transformation** A transformation is a mapping. It is based on a set of rules describing how one (or many) source concepts are transformed into one (or many) target concepts. In this document we consider that transformations are used between different levels of abstraction (e.g. a transformation from functional level to abstract platform level, or a transformation from an abstract platform level to a concrete platform level). Once the mapping rules are defined, a transformation can be automated, e.g. by creation of elements on lower layer.

In order to simplify the transition between levels of abstraction and ensure a coherent timing requirements decomposition the following guidelines are recommended:

- The transition between different levels of abstraction are done with a transformation and this transformation shall implement a one-to-one mapping (one-to-many or many-to-one concepts mapping shall be avoided).
- A consequence of the previous guideline, is that a decomposition is done at a given level of abstraction (decompositions during the transition between levels of abstraction shall be avoided).

These guidelines are illustrated on the following example, which shows a chain of decompositions and transformations from a functional-level description down to a AUTOSAR classic and adaptive platforms description.

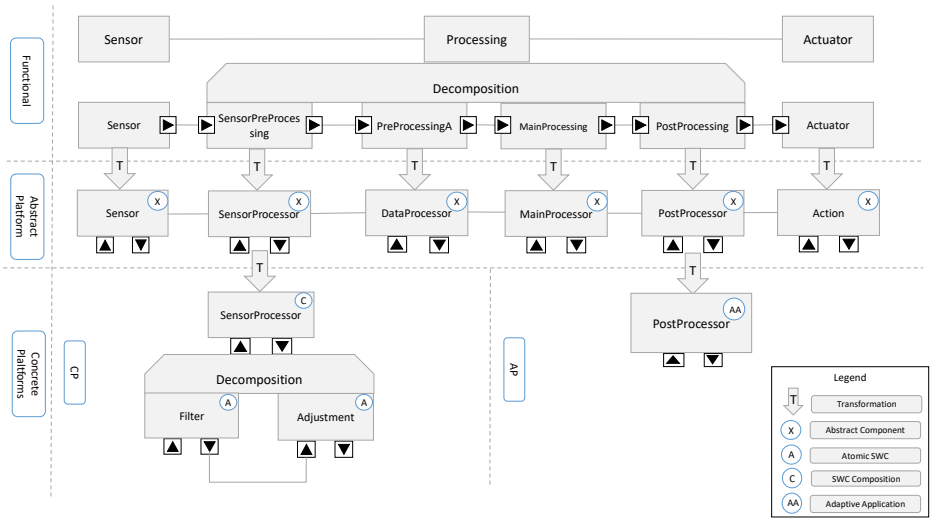

Figure 2.1: Timing Requirements Abstraction Levels and Decompositions

2.2 Basic Concepts of Real Time Architectures

2.2.1 Real Time Architecture Definition

An E/E architecture is the result of early design decisions that are necessary before a group of stakeholders can collaboratively build a system. An architecture defines the constituents (such as components, subsystems, ECUs, functions, compilation units ...) and the relevant relations (such as “calls”, “sends data to”, “synchronizes with”, “uses”, “depends on” ...) among them. In addition to the above-mentioned structural aspects, a real-time architecture shall provide means to fulfill timing requirements. Like for the system’s constituents, real-time architecting consists of decomposing timing requirements and identifying relationships (such as refinement and traceability) among them. In fact, the timing requirements decomposition is a consequence of the structural decomposition where timing requirements are in part inherited by the decomposed units. However, while structural decomposition could be driven by functional concerns, input/output data flows, and/or provided/required services, timing decomposition is a more complex task to achieve. Correct timing requirement decomposition must be locally and globally feasible. Locally each subcomponent’s timing properties must fulfill the assigned timing requirements. The design of a real-time software architecture consists of finding a functional decomposition and a platform configuration whose timing properties allow fulfilling local and global timing requirements.

Timing properties are highly dependent on the underlying software and hardware platform resources. Moreover, access to shared platform resources by the decomposed units introduces some overhead (like blocking times or interferences ...). Timing properties will depend on:

- The chosen *placement* (e.g. allocation of function blocks onto a device, connecting a device to a network);
- The chosen *partitioning* (e.g. assigment (split/group) of entities to schedulable entities);
- The chosen *scheduling* (e.g. priority assignment of schedulable entities or shared resources access protocol).

In order to assess these architectural choices with regard to timing requirements, timing analysis is necessary. Analysis methods and associated timing properties used for such an assessment can depend on the kind of real-time architecture under consideration (e.g. time-triggered or event-triggered architecture). Chapter 9 details this aspect. Timing analysis can be introduced at the system level as a prediction instrument for the refinement of system functions toward their implementation [5]. Although timing analysis in early development phases requires to make assumptions about the resources of the implementation platform, it constitutes a sound guideline for the decomposition and refinement of timing requirements.

From the application point of view the following two timing properties are particularly important:

- *Execution and transmission times*;
- *Response times*.

First introduction of these terms is given below. A more detailed description and classification of these notions is provided in Chapter 9.

2.2.2 Execution and Transmission Times

The execution time of a schedulable entity is the duration taken by the schedulable entity to complete its execution on a computing resource (e.g. ECU). When referring to execution time we mean the net execution time. It only includes the duration a schedulable entity is actually executed on the computing resource. Not included is the time where it may be suspended or preempted due to other schedulable entities sharing the same computing resource, the setup time required to prepare the computing resource on the start or when resuming the execution of a schedulable entity.

Similarly, the transmission time of a signal/message/frame on a communication resource (e.g. bus, network) is the duration taken by the signal/message/frame to transit from its source to its destination without any consideration of other signals/messages/frames transiting on the same communication resource.

An execution/transmission time is a quantitative property that can be described with the following characteristics:

- A *statistical qualifier* (worst, best, mean/average) representing the bounds of execution/transmission time. This bound could be the upper bound which corresponds to the worst-case execution/transmission time (WCET/WCTT), the lower bound corresponding to the best-case execution/transmission time (BCET/BCTT), or the average-case execution/transmission time (ACET/ACTT) which could be useful for performance analysis. Among these three qualifiers, the WCET is the most commonly used for timing properties verification/validation of real-time systems.
- A *method* (estimation (e.g. simulation), measurement, calculation (static analysis)) denoting the way an execution/transmission time is obtained. The precision of an execution time is highly dependent on its source. For instance, input data used for measurements triggers specific branches of the function/program which impacts the measured execution time value. For that reason, measurements can only provide average execution time or a distribution of execution times. To obtain execution time upper bound, static analysis techniques are employed (abstract interpretation, model checking ...).
- An *Accuracy* (see Glossary of Terms). The accuracy of the evaluated WCET/WCTT depends on many factors among which the level of details of the software (instruction level) as well as the level of details of the execution/communication resource (like cache mechanisms). This latter could provide elements of unpredictability like branch prediction mechanisms that could affect the WCET

analysis by making it more complex to achieve and too pessimistic. In order to avoid overdesigning execution platforms, and in order to allow accurate response time analysis (see the following subsection) WCET/WCTT analysis should provide safe but accurate WCETs/WCTTs.

Sometimes, a WCET/WCTT can be a requirement to satisfy, especially at the very low levels of abstraction once the ECUs, network and deployment are fixed. However, in the very upper levels of abstraction, timing requirements usually refer to end-to-end response time bounds defined in the following subsection.

2.2.3 Response Time

The response time of a schedulable entity is the time duration taken by the schedulable entity to complete its execution. Unlike for execution time, the response time takes into account other schedulable entities that are sharing the same execution/-communication resource. Hence, the response time of a schedulable entity comprises its execution time and additional terms induced by the concurrent access to shared resources (blocking times, jitters...). See Chapter 9 for more details.

An end-to-end response time is a response time in which several schedulable entities are involved. These schedulable entities form a chain. First schedulable entity of the chain is called the *source* schedulable entity and the last one is called the *sink* schedulable entity. The end-to-end response time is the elapsed time until the sink schedulable entity of the chain terminates its execution.

Like an execution time, a response time is a quantitative property that can be described with the following characteristics:

- A *statistical qualifier* (worst, best and mean/average). The worst-case response time (WCRT) is the upper bound usually computed by timing analyses to assess timing requirements fulfillment. A more detailed definition of statistical qualifier is given in Chapter 9.
- A *method* (estimation, measurement, calculation (static analysis)) denoting the way a response time is obtained. Methods for response time determination are given in Chapter 9.
- An *Accuracy* (see Glossary of Terms). The accuracy of a WCRT is highly dependent on the accuracy of the Worst Case Executions Times of the executable entities that are involved in the chain.

2.3 Relation between Timing and Data Flow

The functional behavior of a real-time system can not only be affected by its end-to-end timing behavior but also by its internal data flow. A **cause-effect chain** represents the data flow among communicating components, by relating read events of a consumer

component to the corresponding write events of a producer component. Depending on the abstraction level, a component might be a functional block, a software component, or a schedulable entity.

It is important to mention that the specific data flow is observed on job-level. Take Figure 2.2 as an example. It shows a cause-effect chain of three periodic real-time tasks (schedulable entities). The different execution instances of schedulable entities are called **jobs**. With implicit communication, each job reads its input at the beginning of its execution, executes on that data, and writes (publishes) the results after completion. This behavior varies for every job, because the physical execution times of schedulable entities are affected by a multitude of influences. Typical reasons are: input data, execution cycle, hardware platform, and scheduling design. Each dependency on the job level is covered in the cause-effect chain (it can be represented as an "instance" of a cause-effect chain). In the given example in Figure 2.2, every third job of the 5ms task within each 20ms hyperperiod may read from different producer jobs, depending on the response time of the prior jobs in the cause-effect chain. As a result, the data flow is not deterministic.

General remark: An **event chain** (TimingDescriptionEventChain in [2]) in AUTOSAR is used to *specify* the causal relationship between two timing events. An example would be "Event *BrakeAcutated* occurs if, and only if, event *BrakePedalPressed* occurred before" This is not done on job-level but allows to constrain the relation, e.g., with a latency constraint.

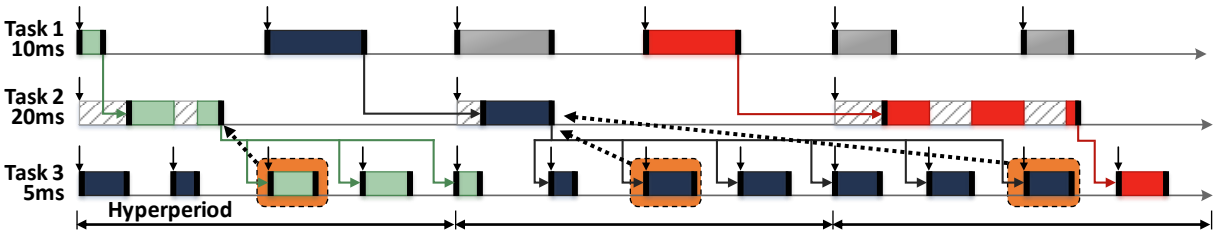

Figure 2.2: Missing data-flow determinism due to execution time variations[6]

How specific the data flow needs to be described depends on the function that shall be implemented. While for some functions it is sufficient to know, that a sensor value will be processed within a given deadline, more complex functions incorporate fork and join points, where different input data shall be combined. An example is given in Figure 2.3. It shows a coarse-grained functional model of a perception and object detection pipeline. Different types of sensors are triggered periodically, while some of the processing is done in a pipelining fashion. The images of a front and a rear camera are combined by an image stitching to generate a 360° image. The Lidar data can be classified in ground points (static objects such as the road, trees or buildings) and non-ground points (moving objects such as vehicles, pedestrians, ...). The non-ground points allow to define regions of interest (ROIs), where the image based classification can provide additional information about the type of objects. Those moving objects are then tracked in the model of the environment.

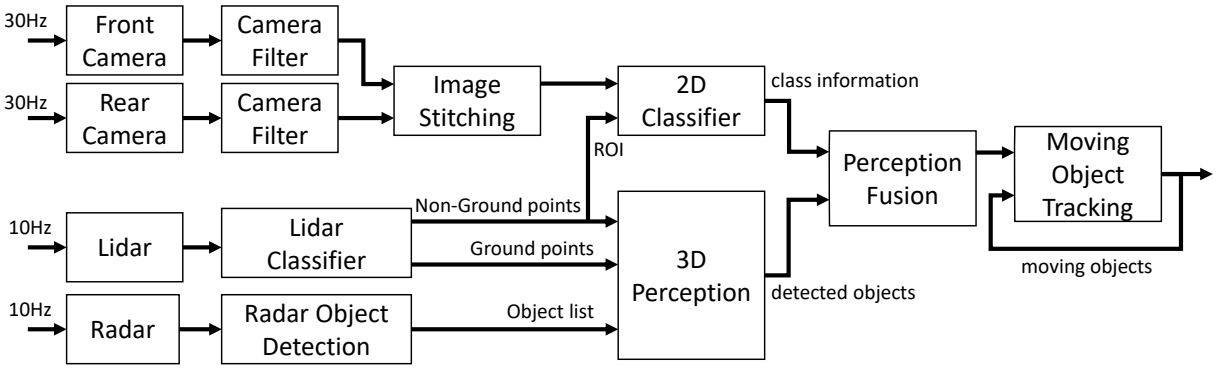

Figure 2.3: Dependent cause-effect chains including data fusion after event-driven processing pipelines

The question is how the data flow within this model can be specified. One approach might be to specify time budgets as upper latency bounds for each processing pipeline. This does not guarantee a deterministic data flow here, since jitter propagates independently in each pipeline path. As a result, the relative data age of the different inputs that shall be combined varies. This *data-age dispersion* becomes problematic e.g., when one path observes a worst-case latency while the other one observes a best-case latency. Consequently, the lower and upper bound for the relative data age can be calculated by comparing the best- and worst-case latency of the two cause-effect chains. Although such a behavior may be sufficient for some functions, it is problematic because:

1. The latency of the individual cause-effect chains likely exceeds the sampling period at the beginning of the chains. As a result, a large data-age dispersion means that the join point combines samples from different sampling times.
2. The data-age dispersion is the result of the different cause-effect chain latencies, which are a result of implementation and run-time jitter and cannot be determined a priori. Such a specification would be beneficial for use-cases where the requirement “both inputs of a functional block have the same data age resp. are originated in the same input sample” should be expressed. Especially when dealing with a larger number of cause-effect chains, this becomes unintuitive since the dispersion depends not only on the longest latency but also on the shortest one.
3. The behavior of each cause-effect chain is implementation dependent and may change with every modification. Especially correlations between the latencies of cause-effect chains are the result of a specific implementation and can not be covered in the model.

As soon as the cause-effect chains comprise loops, like the tracking of moving objects, the problem is amplified. Combining an output data sample, which originates in an uncertain data-age dispersion, again with new input data of uncertain data-age dispersion accumulates the non-determinism.

There are different options how such a model can be further constrained to reduce the non-determinism. The first important aspect is to define communication semantics and execution order constraints. For the model in Figure 2.3, one might ask the questions:

1. What is the **communication semantic** of an edge between two components? Does it mean:
   (a) **FIFO semantics**: Buffering, optionally with maximum queue size (non-destructive write, destructive read)?
   (b) **Register semantics**: Only the most recent data sample is available and can be read multiple times (destructive write, non-destructive read)?
   (c) **other semantics** such as destructive-write/destructive-read or last-in-first-out (stack)? What is applicable depends on the intended function.
2. When is a component **activated**:
   (a) **Time-based** with a given period and offset?
   (b) **Event-based** and on what kind of event? A particular trigger input, or when all input data is available, or when at least one input is available, or a combination of inputs, or...?
3. What is the **input-output relation** of a component? Take as an example the 2D classifier:
   (a) Is it forced to produce an output at all, or may it internally decide if an output is needed?
   (b) If camera frames are available with 30Hz and a new ROI arrives with 10Hz, does it produce an output for every ROI or for every camera frame?

Depending on how those questions are answered, it is possible to enforce a *deterministic data flow*. As a result of determinism, confidence in the functional behavior can be increased and costs for testing and validation decreased. There are different approaches available how to specify a deterministic data flow, either directly by using data flow graphs (e.g., synchronous data flow graphs (SDFs), which have their specific semantics), or indirectly by using time programming paradigms that combine a timing specification with specific communication semantics. One example, the Logical Execution Time (LET) paradigm, is described in the following Section 2.3.1.

2.3.1 Logical Execution Time and data-flow determinism

Response times are statistical qualifiers in the sense of Section 2.2.3, meaning that the time that a schedulable entity takes for completion may jitter. Consequently, the times when schedulable entities read and write data may also jitter. A cause-effect chain is a chain in which the outcome of one entity influences the behavior of the next entity in the chain. The motivation of a *logical timing abstraction* is the observation that the relevant behavior of real-time programs is determined by the data flow within a cause-effect chain. For this data flow, it is only important when inputs are read and outputs are written, and not when programs complete code execution.

The concept of a cause-effect chain allows specifying a unique data flow between the entities of a chain. Such a unique data flow is called a deterministic data flow and the related property is data-flow determinism. Data-flow determinism guarantees that the behavior of a chain is preserved under any platform execution. Data-flow determinism is an important basic concept of real-time architectures, because it increases design predictability, robustness, and portability.

While data-flow determinism is an established concept in event-driven chains, where an output triggers the execution of the next entity in a chain, data flow is generally not preserved in cause-effect chains where the entities are triggered periodically (time based activation).

One way to guarantee data-flow determinism is to specify the timing of read and write operations of an entity in a chain. This principle is the basis of the Logical Execution Time (LET) paradigm. Despite its name, LET can be applied to both computation and communication . Unlike the response time described in the previous section 2.2.3, with LET the response time and data flow is fixed and independent of the actual scheduling behavior. This allows simpler specification and implementation of deterministic end-to-end behavior of cause-effect chains.

Figure 2.2 already shows an example of the missing data-flow determinism when response times are considered as a baseline to define the availability of outputs. As a consequence, the latency of a cause-effect chain which includes these schedulable entities, can not be composed as a simple addition of timing budgets, nor can the timing requirements for this behavior be simply decomposed from high level requirements (see Section 3.1). Modifications in such designs likely affects the data flow and a reuse of existing components requires significant analysis/testing effort. Such situations typically occur in multi-core designs.

In contrast, LET provides a *timing abstraction of the physical execution* of a real-time program. This is accomplished by specifying explicit points in time when input data is read (release point) and output data is published/written (termination point). The *constant time* interval in-between is referred to as the LET. When the actual physical execution terminates, the outputs are not made immediately available but are published at the end of the LET (e.g. with means of buffering). Reading inputs and publishing outputs takes place at the LET events in logically zero time. Figure 2.4 shows this relation between the physical execution and a logical timing abstraction in detail.

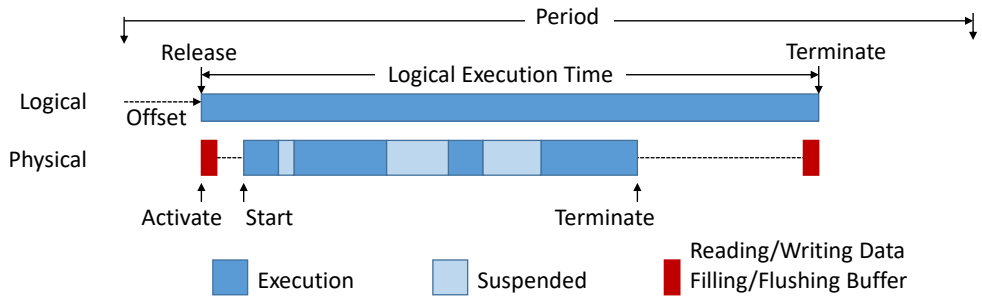

Figure 2.4: Logical Execution Time Abstraction

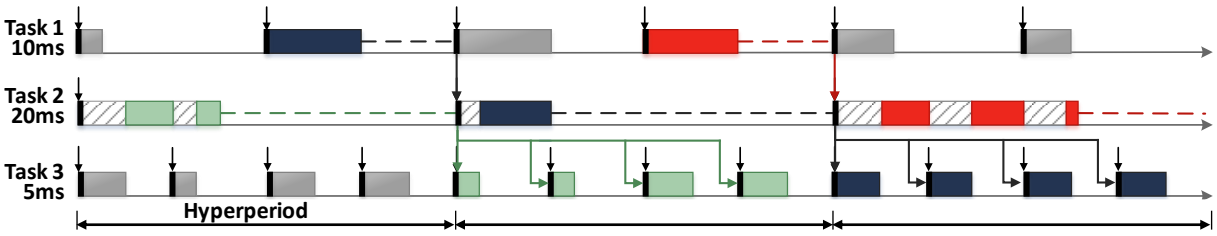

Figure 2.5: Deterministic data flow with LET [6]

As a consequence, LET enforces a deterministic input/output timing with a constant response time and zero jitter. Figure 2.5 provides an example of the data flow when LET is applied to the cause effect chain from Figure 2.2. The new data flow depends only on the LET events and is not affected by the physical execution timing. In Figure 2.5, the duration of tasks' LET intervals is assigned equal to their periods. However, LET does not imply such a constraint and tighter LETs ($LET < period$) as well as offsets can be used to reduce the end-to-end latency. On the other hand, overdimensioning of the duration of an LET interval can be used as a design instrument to improve data-flow robustness, in case that future system modifications affect the WCRT negatively (reducing the margin between WCRT and LET interval length). Of course this is a trade-off with the total end-to-end latency of the cause-effect chain.

Even though not explicitly discussed nor even mentioned in the original LET literature [7] [8], there is a conceptual bound of $LET \leq period$. Beyond that bound, overlapping execution is possible and some of the fundamental assumptions of the classical LET including data-flow determinism do not hold any more. This validity bound is essential, because of two aspects. First, the specification of LET intervals (by means of AUTOSAR Timing Extensions for the classic platform) does not enforce such a bound and it is crucial because in practical AUTOSAR applications, $LET > period$ is an important use case. As an example, the runnables that are involved in a cause-effect chain may be deployed to different ECUs, extending the problem of deterministic data flow to distributed systems with significant communication delays. Second, the concept of a logical timing abstraction can also be beneficially used as a specification technique in early stages of the design. On the functional level for example, this may cover the composition and decomposition of LET intervals that are used to specify the data flow among multiple functional blocks. Again, it is likely that a end-to-end latency of a composition of functional blocks exceeds the triggering period. To close that gap, an extension of the LET semantics is required.

2.4 Languages for Timing Requirements Specification

The AUTOSAR methodology is based on the AUTOSAR language and its timing extensions. AUTOSAR is the language for the software implementation levels but not applicable at the functional levels (analysis and design). Therefore, in order to ensure a complete model-based approach for timing requirements decomposition, a complementary modeling language for functional levels has to be used. EAST-ADL2 [9] and its timing extension TADL2 [10] allow functional levels specification with precise timing

models. Moreover, TADL2 and AUTOSAR Timing extensions are sharing the same base concepts which may facilitate the translation of timing requirements from the functional level to the AUTOSAR level (where timing requirements are expressed with TIMEX).

Therefore, EAST-ADL / TADL is briefly presented as an example of modeling language for the support of the functional levels of a methodology for timing requirements decomposition.

2.4.1 EAST-ADL / TADL

EAST-ADL is an Architecture Description Language (ADL) for automotive embedded systems, developed in several European research projects. It is designed to complement AUTOSAR with descriptions at higher level of abstractions. Aspects covered by EAST-ADL include vehicle features, functions, requirements, variability, software components, hardware components and communication.

TADL2 (Timing Augmented Description Language) language concepts can be used in specific steps of the GMP (Generic Methodology Pattern, see section 3.3.2) methodology to describe timing information. TADL2 allows the specification of timing constraints that may express the following timing properties/requirements:

- Execution Time (Worst-case, Best-case, Simulated, Measured)
- End-to-end Latency
- Sampling Rates
- Time Budget
- Response Time
- Communication Delay
- Slack
- Repetition pattern
- Synchronization
- ...

TADL2 base concepts are quite equivalent to those of AUTOSAR TIMEX presented in the following section.

2.4.2 Basic concepts of AUTOSAR TIMEX

According to [2], the primary purpose of the timing extensions is to support constructing embedded real-time systems that satisfy given timing requirements and to perform timing analysis/validations of those systems once they have been built.

The AUTOSAR Timing Extensions provide a timing model as specification basis for a contract based development process, in which the development is carried out by different organizations in different locations and time frames. The constraints entered in the early phase of the project (when corresponding solutions are not developed yet) shall be seen as extra-functional requirements agreed upon by the development partners.

This way the timing specification supports a top-down design methodology. However, due to the fact that a pure top-down design is not feasible in most of the cases (e.g. because of legacy code), the timing specification allows the bottom-up design methodology as well.

The resulting overall specification (AUTOSAR Model and Timing Extensions) shall enable the analysis of a system's timing behavior and the validation of the analysis results against timing constraints. Thus, timing properties required for the analysis must be contained in the timing augmented system model (such as the priority of a task, the activation behavior of an interrupt, the sender timing of a PDU and frame etc.). Such timing properties can be found all across AUTOSAR. For example the System Template provides means to configure and specify the timing behavior of the communication stack. Furthermore the execution time of executable entities can be specified. In addition, the overall specification must provide means to describe timing constraints. A timing constraint defines a restriction for the timing behavior of the system (e.g. bounding the maximum latency from sensor sampling to actuator access).

Timing constraints are added to the system model using the AUTOSAR Timing Extensions. Constraints, together with the result of timing analysis, are considered during the validation of a system's timing behavior, when a nominal/actual value comparison is performed.

The AUTOSAR Timing Extensions provide some basic means to describe and specify timing information: timing descriptions, expressed by events and event chains, and timing constraints that are imposed on these events and event chains. Both means, timing descriptions and timing constraints, are organized in timing views for specific purposes. By and large, the Timing Extensions serve two different purposes. The first is to provide timing requirements that guide the construction of systems which eventually will satisfy those timing requirements. The second purpose is to provide sufficient timing information to analyze and validate the temporal behavior of a system.

The remainder of this section describes the main concepts defined in the AUTOSAR Timing Extensions.

2.4.2.1 TIMEX Work Products

The following part decribes the different TIMEX Work Products to provide a general overview on them. Further, much more detailed descriptions are given in [2] Chapter 2 (Timing Extensions Overview)

Events. The notion of Event is used to describe that specific observable events occur in a system and also at which locations in this system the occurrences are observed. These are related to predefined Event types and are used to specify different actions (eg. Read/Write data to ports, Send/Receive data via network, Start/Terminate executables ...).

Event Chains specify a causal relationship between two Events. For example Event B occurs if and only if Event A occurred before.

Timing Constraints imposed on Events. Event Triggering Constraint imposes a constraint on the occurrences of an Event in a temporal space (periodic, sporadic, specific pattern).

Timing Constraints imposed on Event Chains. Event Triggering Constraints are used to specify a reaction or age, e.g the maximum distance of two following events. Latency and synchronization timing constraints specify that a stimuli or response event must occur within a given time interval (tolerance) to be said to occur simultaneous and synchronous respectively.

Additional Timing Constraints. AUTOSAR Timing Extensions provide Timing Constraints which are imposed on Executable Entities, namely the Execution Order Constraint and Execution Time Constraint.

3 Timing Requirements on Design Levels

The decomposition of timing requirements is a primary concern for the design and analysis of a real-time system. At the beginning of the system design process, timing requirements are expressed at the level of the customer functionality identified in the specification. The development of the customer functionality requires its decomposition into small and manageable components. This decomposition activity called architecting implies also a decomposition of timing requirements attached to the decomposed functionality. This chapter gives an overview of the proposed approach for the decomposition of timing requirements.

3.1 Timing Requirements Decomposition Problem

Mastering timing requirements is one key success factor for the development and integration of state of the art automotive E/E-systems. Timing requirements should be monitored continuously during the complex development process of a vehicle, and further shall be reused and communicated for the re-use of functions or components to other vehicle projects: timing requirements have to be described systematically and carefully. The required level of detail can vary from timing constraints for high level customer related features at the vehicle level, over timing requirements for the control of a power amplifier for a particular actuator, to ECU-internal timing for data synchronicity of software functions on a multi-core microcontroller at the operational level.

As illustrated in Figure 3.1, the development process follows the well-known V-model, which describes a systematic and staggered top-down approach from system specifications to system integration. On the left branch process steps of specification are described, implementing decomposition from an entire E/E-system to single components. The base of the V describes implementation and associated test procedures. Following the right branch of the V testing and integration procedures up to vehicle system integration can be read in bottom up order.

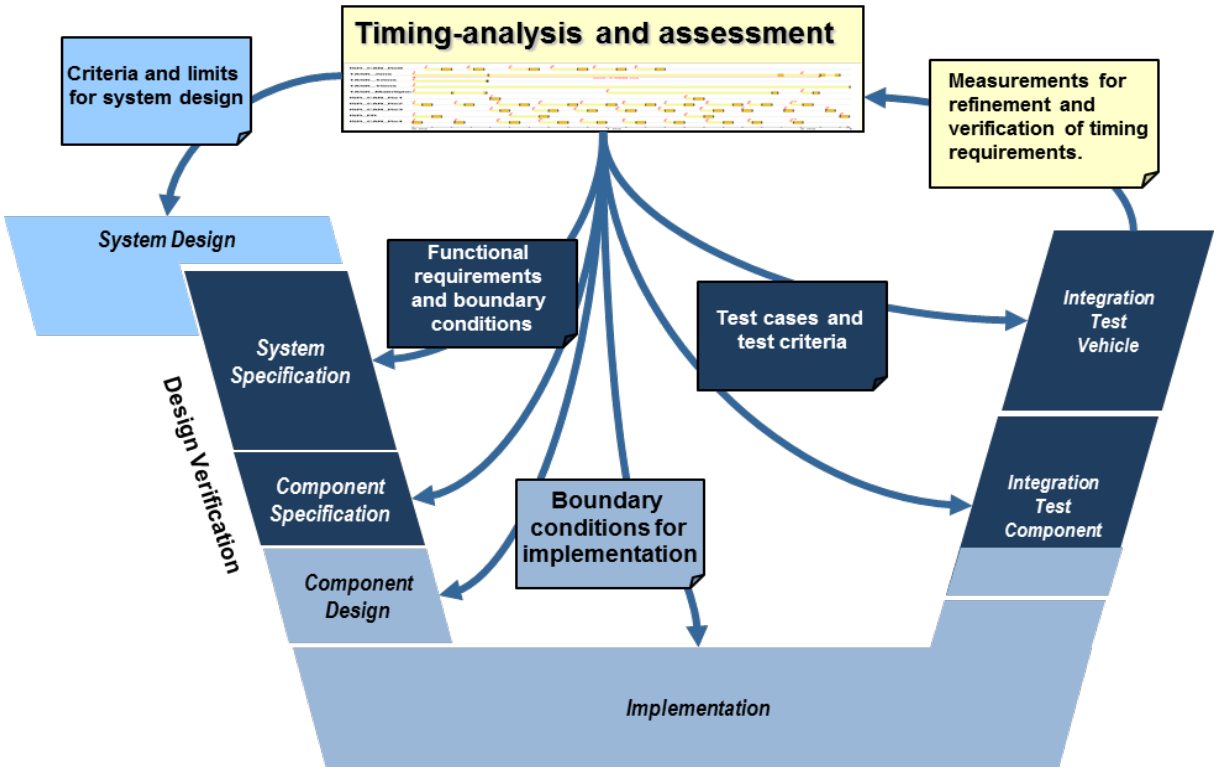

Figure 3.1: Application of timing analysis in a development process according to the V-model

According to these basic steps of an automotive OEM development process, requirements shall be traceable in any process step. This means that timing requirements shall be identifiable and traceable from a requirements specification via a supplier's performance specification to a test and integration documentation (protocols). As far as E/E-processes are concerned this means that timing requirements shall resist the process transformation between two companies like OEM an tier1-supplier and further down to tier2 and 3 suppliers. This can only be achieved by using a standardized system of description and methodology, referencing the model artifacts that are generally exchanged between development partners.

The AUTOSAR Timing Extensions (TIMEX) [2] based on the AUTOSAR System Template, represents the standardized format for exchange of a system description within an AUTOSAR compliant software development process. In addition TIMEX is an optional component which does not imply changes in the AUTOSAR System Template. The concept of the observable event, which occurs or can be observed in a referenced modeling artifact e.g. a RTE-port, allows specifying observation points and sequences of events in causal order (event chains) with additional timing constraints on them. The TIMEX concept is assumed to meet all use-cases of describing temporal behavior in an AUTOSAR system by means of timing requirements.

Unfortunately the OEM development process does not start with AUTOSAR. AUTOSAR only represents an implementation view for some software components, but not a view on higher level functional concepts that can comprise non software

functions. Currently requirements are described in natural language at the very beginning of the process. These requirements have to be “formalized” in a non-natural language in order to assess them and allow their decomposition. The assessment of timing requirements should be done as early as possible in the development process. To enable this at system/functional level, a system/functional modeling language is needed. This language must provide concepts for functions design modeling and must also provide a formal way to capture and decompose timing requirements during the functional design.

Several approaches based on Architecture Description Languages (ADLs) could be used to fill the gap between requirements specification in natural language and the implementation phase modeled in AUTOSAR. We can cite UML-based [11] Architecture Description Languages: SysML [12] (UML specialization for System Modeling) and MARTE [13] (UML specialization for Modeling and Analysis of Real-Time end Embedded systems). Other approaches that are more domain specific like AADL [14] for aerospace or EAST-ADL [9] for automotive also exist. The choice of the appropriate system/functional level modeling language depends on the internal OEMs’ processes. However, there are some general timing related criteria that are important to consider:

- A support for hierarchical timing requirements process;
- The ease of mapping the decomposed timing requirements to AUTOSAR TIMEX model artifacts that constitutes today the exchange format between the OEM and its suppliers.

In the process of *decomposing* timing requirements, particularly the timing of dataflow is challenging. Achieving a *deterministic* timing along multiple cause-effect chains, such that the desired functional behavior is retained, requires special care in the design process. While in higher level specification, timing can be coarsely decomposed into timing budgets for different (sub)functions, this becomes challenging on the implementation level of AUTOSAR, if not sufficiently well-structured. As an example, consider the initial timing decomposition shown in Figure 1.5 for the yellow and the red cause-effect chain. These budgets can be refined and hierarchically decomposed along the V-model process from Figure 3.1. However, at the implementation level of AUTOSAR this reaches a point, where timing is not generally composable. Meaning that if the behavior of a cause-effect chain, among others factors, depends on the timing of the data, no simple “composition” like adding maximum response-times is possible. This is already illustrated in the example from Figure 2.2: Due to the implementation dependent response time including a possible response time jitter at run time, it remains open from which previous job the third job of the 5ms task within the 20ms hyperperiod may read data. Even worse, this can change with any software update and with hardware operation, such as thermal processor management. As a consequence, the latency of a cause-effect which includes this task, can not be composed as a simple addition of timing budgets (cf. Figure 1.5), nor can the timing requirements for this behavior be simply decomposed from high level requirements. Abstractions such as

SL-LET, supported by suitable implementation of basic software, RTE and OS, however can avoid this “decomposition gap”, such that timing of cause-effect chains can be designed composable.

In the following section an approach based on all these ideas and concepts is drawn which shall give orientation to implement a hierarchical timing requirements process in the own organization and also, in the end, enables the exchange of AUTOSAR TIMEX compliant model artifacts.

3.2 Hierarchical Timing Description

During the early design phase of an automotive development process the architecture discussion is about high level customer related functions. These functions can be detailed in functional “cause and effect” or “activity” chains, which from a temporal view can be budgeted - justified by customer’s experience. The functional quality and thus technical effort dedicated to the customer’s experience is a business decision of a company.

In the AFS example this is the reaction time from the detection by the yaw rate sensor to the motor control of the steering system. This avoids instable behavior during driving in curves.

An other example is a powertrain or chassis control function which can cause inconveniences like bucking during shifting or braking.

From methodological and technical view timing analysis is a tool to assure the desired temporal behavior during the mapping of a functional network to a component network as depicted on Figure 3.2.

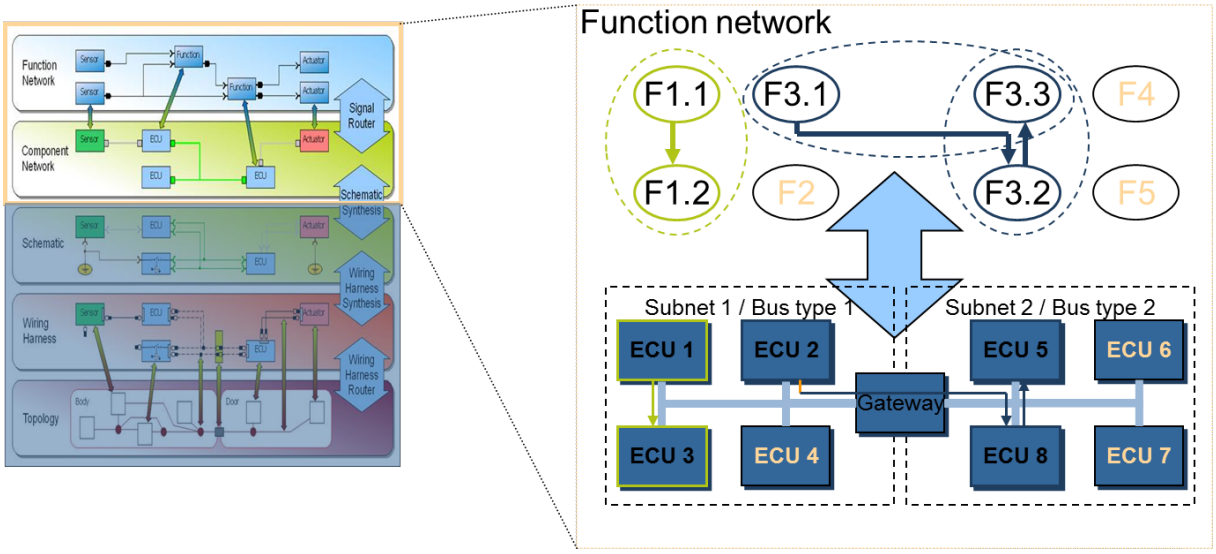

Figure 3.2: Mapping of a function network to a component network

Once the major timing budgets for customer related functions are defined and a distribution of functional parts to hardware components is done [1], a more detailed temporal view of a networking architecture can be made. This allows a first assessment of the feasibility of the function distribution in terms of performance and timing. This process can iteratively be refined during further process steps to obtain more precise analysis results.

For further understanding, it can be assumed that each function in Figure 3.3 is contained in the compositional scope of an AUTOSAR SW-C, where it is represented as an AUTOSAR runnable entity, shortly often named “Runnable”. Other mapping strategies can also be considered. Regardless of the chosen strategy, the mapping is usually constrained by the functional design choices made at the functional level for timing requirements assessment. For instance, a feasibility test based on the computation of the load (utilization) of each hardware resource (ECUs, buses), is based on a given allocation of functions on hardware resources. This allocation has to be taken into account for the mapping of functions to AUTOSAR SW-Cs in order to avoid the mapping of two functions that are allocated on distinct ECUs on the same AUTOSAR SW-C.

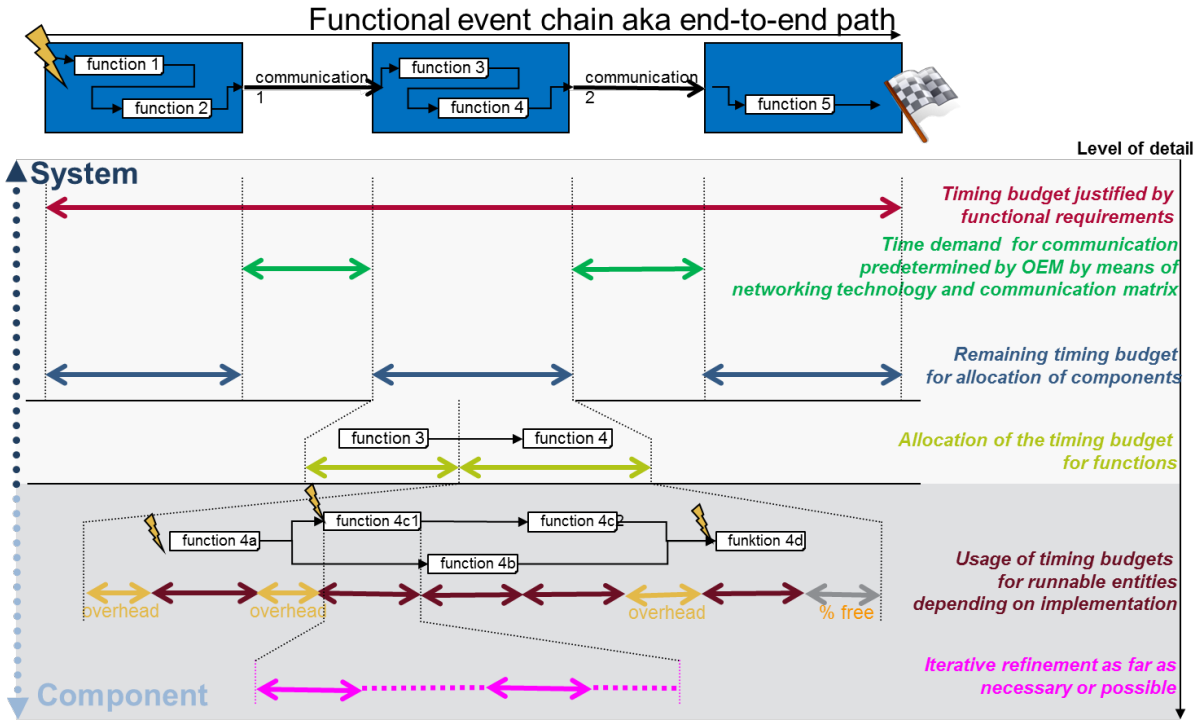

Figure 3.3: Iterative and hierarchical top down budgeting of timing requirements corresponding to response times

Moreover, in many cases timing demands of physical processes, e.g. the start-up and transient oscillation behavior of electrical actuators, consume more than a

[1] In an AUTOSAR development process a software component (SW-C) is defined with a scope local to the hardware component it is mapped on. It contains a functional contribution to the vehicle function with a system wide scope.

few microseconds and thus have to be considered carefully. In a first step the overall timing budget can be split in component-internal and networking parts. As soon as the entire network communication and the type of network are known, the WCRT-analysis of a network can quantify the worst case timing demand for network communication. As shown in the picture above, this divides the overall timing budget in networking budgets and timing budgets for allocation in components (usually ECUs).

This can be enough for an OEM if the development and integration of the component is entirely done by a supplier. In practice a more detailed view considering the timing behavior of a basic software stack and the functions itself is required. Likewise functional relations are more complex, which induces a more complex analysis. During further analysis steps the end to end timing path or chain of functions can be refined following the concepts of Figure 3.3. In the following section we introduce methodologies that provide support for the general process described.

3.3 Methodologies for Timing Requirements Decomposition

As previously stated, the AUTOSAR methodology covers the implementation phase of the process of E/E systems development. However, timing requirements are introduced at the very beginning of the development cycle in the form of textual descriptions by OEMs. An extension of the AUTOSAR methodology is then needed to cover the system/functional architecture design phases where the first functional decompositions and timing requirements decomposition must occur. In fact, one of the most challenging activities in the development of systems is determining a system's dimensioning in early phases of the development - and the most difficult one is the phase before transitioning from the functional domain to the hard and software domain.

Primarily, two questions must be answered. Firstly, how much bandwidth shall the networks provide in order to ensure proper and timely transmission of data between electronic control units; and secondly, how much processing performance is required on an electronic control unit to process the received data and to execute the corresponding functions. As a matter of fact, these questions can only be completely answered when the system is implemented, including a mapping of signals to network frames and first implementations of functions that are executed on the electronic control units. The reason for this is that one needs to know how many bits per second have to be transmitted and how many instructions shall be executed.

An important aspect that impacts the decisions taken during the task of specifying system dimensions is timing. Especially, information about data transmission periods, execution rates of functions, as well as tolerated latencies and required response times provide a framework for performing a first approximation of network and ECU dimensions. This framework allows to continuously refine the system dimensioning during system development when more details about the system's implementation are becoming available. The basic idea is to abstract from operational parameters obtained during the implementation phase, like for example measured or

simulated execution times of functions, and use them on higher levels of abstraction respectively earlier development phases. And, for new functions as a workaround for missing execution time, an activity called Time Budgeting allows the specification of so called time budgets to functions.

The remainder of this section defines the levels that will be considered for timing requirements decomposition. Then, some generic methodological guidelines will be given for conducting timing requirements refinement between these levels.

3.3.1 Functional and Software Architectures Modeling Levels

Prior to the AUTOSAR software architecture levels, we can consider two functional architecture modeling levels defined in [9] that are of interest for timing requirements:

- The *Functional Analysis level* which is centered on a logical representation of the system's functional units to be developed. Typically based on the inputs of automatic control engineering, system design at this level refines the vehicle level system feature specification by identifying the individual functional units necessary for system boundary (e.g., sensing and actuating functions for the interaction with electromechanical subsystems) and internal computation (e.g. feedback control functions for regulating the dynamics of these subsystems). The design focuses on the abstract functional logic, while abstracting any SW/HW based implementation details. Through an analysis level system model, such abstract functional units are defined and linked to the corresponding specifications of requirements (which are either satisfied or emergent) as well as the corresponding verification and validation cases.
- The *Function Design level* provides a logical representation of the system functional units that are now structured for their realizations through computer hardware and software. It refines the analysis level model by capturing the bindings of system functions to I/O devices, basic software, operating systems, communication systems, memories and processing units, and other hardware devices. Again, through a design level system model, the system functions, together with the expected software and hardware resources for their realizations, are defined and linked to the corresponding specifications of requirements (which are either satisfied or emergent) as well as the corresponding verification and validation cases. Moreover, the creation of an explicit design level system model promotes efficient and reusable architectures, i.e. sets of (structured) HW/SW components and their interfaces, hardware architecture, for different functions. The architecture must satisfy the constraints of a particular development project in automotive series production.

The AUTOSAR methodology (see [1] for a general introduction) provides several well defined process steps, and furthermore artifacts that are provided or needed by these steps. Figure 3.4 provides a simplified overview of the AUTOSAR methodology, using the Software & Systems Process Engineering Metamodel notation (SPEM) [4], focusing on the process phases which are of interest for the use of the timing extensions. These represented steps and artifacts are grouped by boundaries in the five following views:

- *VfbTiming* deals with timing information related to the interaction of `SwComponentType`s at VFB level.
- *SwcTiming* deals with timing information related to the `SwcInternalBehavior` of `AtomicSwComponentType`s.
- *SystemTiming* deals with timing information related to a System, utilizing information about topology, software deployment, and signal mapping.
- *BswModuleTiming* deals with timing information related to the `BswInternalBehavior` of a single `BswModuleDescription`.
- *EcuTiming* deals with timing information related to the `EcucValueCollection`, particularly with the `EcucModuleConfigurationValues`.

Further details of these timing views are given in [2]. For each of these views a special focus of timing specification can be applied, depending on the availability of necessary information, the role a certain artifact is playing and the development phase, which is associated with the view.

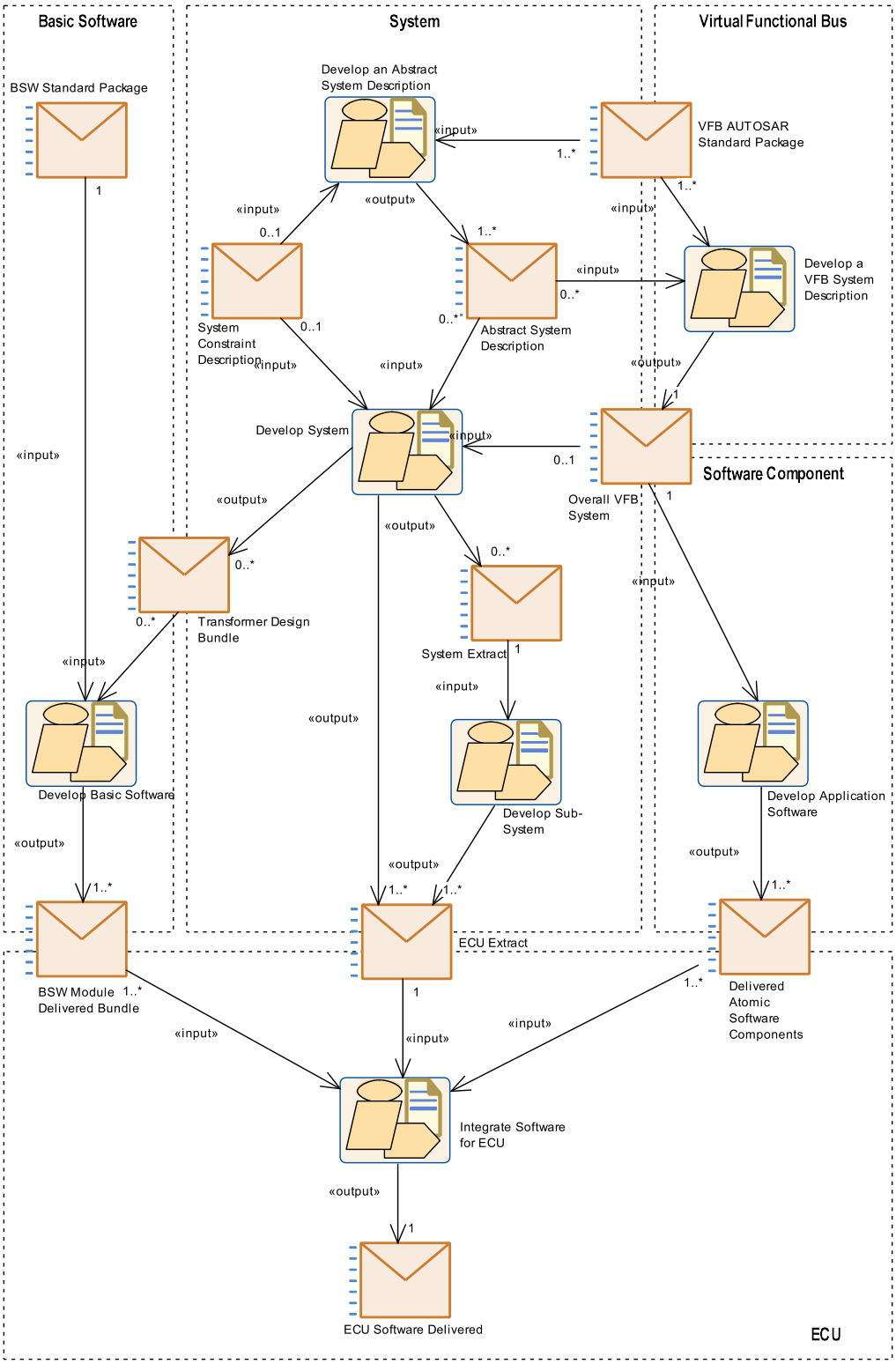

Figure 3.4: SPEM Process model from AUTOSAR Methodology for system design process

3.3.2 Guidelines for Timing Requirements Decomposition

The Generic Methodology Pattern (GMP) developed in the TIMMO-2-USE project [15] is an example of a process that defines generic steps for timing requirements refinement. Theoretically, those generic steps are applicable at every level defined in the previous section (including the AUTOSAR levels). Basically, at each abstraction level, GMP takes as input timing requirements and after a sequence of steps gives as output refined timing requirements. GMP defines six main steps. Some of them have been merged in the following short description:

- *Step1 - Create Solution*: describes the definition of the architecture without any timing information. This step can consist in a refinement of an already existing architecture coming from the upper level. Timing requirements shall guide the creation or revision of a solution.
- *Step2 - Attach Timing Requirements to Solution*: describes the formulation of timing requirements in terms of the current architecture. This can imply a transformation of timing requirements coming from the previous level, in order to be compliant with the timing model of the current level of abstraction. For instance in the AUTOSAR SwcTiming view a timing requirement can be modeled with a timing constraint attached to events or event chains.
- *Step 3 - Create, Analyze and Verify Timing Model*: describes the definition of a formalized model for the calculation of specific timing properties of the current architecture. In this step relevant timing analysis methods can be applied to verify timing requirements against calculated timing properties (e.g. maximal load for a bus). If timing requirements are not verified by timing properties resulting from the analysis, the previous steps shall be iterated until a satisfactory solution is found.
- *Step 4 - Specify and Validate Timing Requirements*: describes the identification of mandatory timing properties and their promotion to timing requirements for the next level.

Chapter 9 contains timing properties and methods of interest for each use-cases described in chapter 4, chapter 5, chapter 6 and chapter 7 to ensure correct timing requirements decomposition.

Timing constraints are added to the system model using the AUTOSAR Timing Extensions. Constraints, together with the result of timing analysis, are considered during the validation of a system's timing behavior, when a nominal/actual value comparison is performed.

3.4 Conclusions

To apply timing requirements decomposition in a comprehensive way several conditions have to be fulfilled:

- All basic terms shall be unified. This means a term like WCRT has the same meaning and comprehensive understanding all over the industry.
- The structure of describing timing aspects shall be unified. For this need AUTOSAR TIMEX delivers an appropriate approach for the implementation driven perspective of AUTOSAR. It does not apply to higher levels of abstraction, because as soon as no AUTOSAR concepts like Software Components and Runnable exist, there is no meaning.
- The methodological approach for introducing timing analysis in a timing aware development process shall not be reduced to the definition of TIMEX artifacts referring to AUTOSAR system template artifacts. Additionally information of higher abstraction levels in earlier design phases shall be transferred to AUTOSAR modeling without loosing exactness. This requires reference points valid within all phases and levels of abstraction.
- The methodology shall meet the needs of large scale organizations. This means the methodology shall be applicable tailor-made to the processes ruling a particular large scale organization.

The elements presented in this chapter allow a formal timing requirement decomposition described in the top level active steering example introduced in Chapter 1.

4 Timing on Functional Level

4.1 Introduction

Functional timing is specified at the Functional Level as defined in section 2.1.1. It consists in specifying timing requirements on an abstract architecture of the system functions. As illustrated on figure 4.1, two kinds of model elements are needed:

- **Functional Architecture Model Elements** are modeling concepts to capture the vehicle functions, their inputs, their outputs and their connections.
- **Functional Timing Model Elements** are modeling concepts to capture timing requirements on observable elements of the functional architecture. In order to be consistent with AUTOSAR timing extensions, these timing model elements make references to the functional architecture model elements.

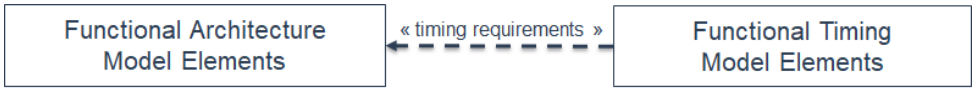

Figure 4.1: Functional Architecture Models and Functional Timing Models

The following sections presents key model elements for the Functional Architecture and the Functional Timing.

4.1.1 Functional Architecture Model Elements

To capture the functional architecture, some generic model elements are needed. To support the definition of the Functional Architecture Model, Figure 4.2 gives an informal metamodel of these model elements and their relationships.

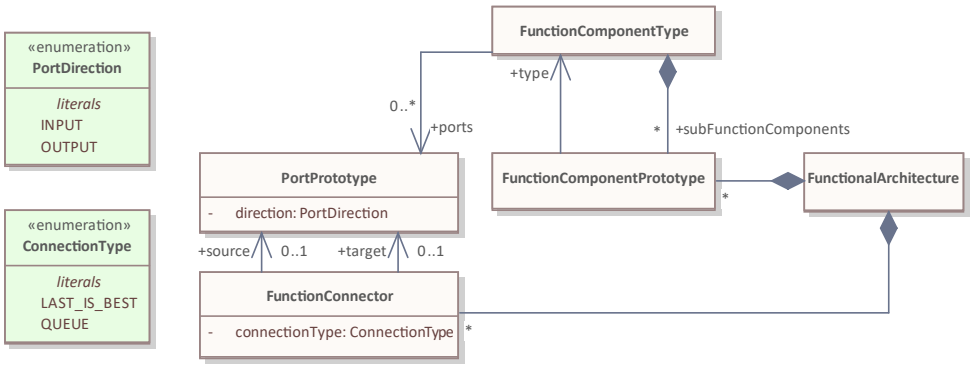

Figure 4.2: Generic Functional Architecture Concepts

- A **Vehicle** provides a set of high-level **vehicle functions**.
- A vehicle function can be decomposed into **FunctionComponentType**s. A FunctionComponentType can be shared between different vehicle functions. In other

words, a FunctionComponentType can contribute to the realization of different vehicle functions.

- A **FunctionComponentType** is equivalent to a *ComponentType* in Autosar. It defines a function type that can be decomposed into **subFunctionComponents**. Like in Autosar, this decomposition is achieved in two steps: (1) a FunctionComponentType of the subfunction is created, and (2) a **FunctionComponentPrototype** corresponding to the role played by the subfunction is created in the composite FunctionComponentType. A **FunctionComponentType** can have connection points called **PortPrototype**s.
- A **FunctionConnector** is a connection link connecting the **PortPrototype**s of **FunctionComponentTypes**. A connection semantic may be defined, as either **Last-is-best** or **Queue**.
- A **PortPrototype** defines the input or output of a **FunctionComponentType**.
- A **FunctionalArchitecture** is recursively composed of **FunctionComponentPrototype**s and their connections.

4.1.2 Functional Timing Model Elements

A timing model at the functional level consists in specifying timing constraints and budgets by referencing the PortPrototypes in the functional model.

Timing constraints can be related to either one or two PortPrototypes in the functional model. Constraining a single PortPrototype can be used to define a **period**, **frequency** or **inter arrival time** of one repeating event.

By relating two PortPrototypes with a timing constraint, it is possible to define **latency timing constraints**. Examples are:

- A **Function response time budget** is a timing budget for gross response time.
- A **Function connector communication time budget** is a timing budget for gross communication time.
- An **End-to-end time budget** is a combination of **function response time budgets** and **function connector communication time budgets** in a chain of connected **FunctionComponentType**s.

4.1.3 From Functional Level to Autosar

This section presents transitions from Timed Functional Architectures to Autosar abstract, classic and adaptive platforms.

- The transition from the timed functional architecture to the Autosar abstract platform is done through a one-to-one transformation from FunctionComponentPrototypes to Autosar Abstract Components. Timing constraints are mapped using TIMEX.
- The transition from the timed functional architecture to the Autosar classic platform is done through a one-to-one transformation from FunctionComponentPrototypes to Autosar Software Components. Timing constraints are mapped using TIMEX.
- The transition from the timed functional architecture to the Autosar adaptive platform is done through a one-to-one transformation from FunctionComponentPrototypes to Autosar Adaptive Applications. Timing constraints are mapped using TIMEX.

4.1.4 Functional Modeling Languages

Table 4.1 presents the mapping of functional timing concepts presented in sections 4.1.1 and 4.1.2 to some existing modeling languages that can be used for functional modeling.

Functional Timing Concept	*EAST-ADL2 / TADL2 Concept*	*UML / SysML / MARTE Concept*
Vehicle Function	EAST-ADL:: Structure:: FeatureModeling:: Feature	UML:: Class or SysML:: Block, or a specific Feature stereotype.
FunctionComponentType	EAST-ADL ::Structure:: FunctionModeling ::AnalysisFunctionType or EAST-ADL:: Structure:: FunctionModeling:: DesignFunctionType	Class, Block, or a specific FunctionType stereotype.
FunctionComponentPrototype	EAST-ADL:: Structure:: FunctionModeling:: AnalysisFunctionPrototype or EAST-ADL:: Structure:: FunctionModeling:: DesignFunctionPrototype	UML:: Property or SysML:: Part
PortPrototype	instanceRefs of EAST-ADL:: Structure:: FunctionModeling:: FunctionFlowPort or EAST-ADL:: Structure:: FunctionModeling:: FunctionClientServerPort	UML:: Port or SysML:: FullPort
FunctionConnector	EAST-ADL:: Structure:: FunctionModeling:: FunctionConnector	UML:: Connector
FunctionalArchitecture	EAST-ADL:: Structure:: SystemModeling:: SystemModel	UML:: Class or SysML:: Block or specific FunctionalArchitecture stereotype.

Function response time budget	EAST-ADL:: Timing:: TimingConstraints:: ExecutionTimeConstraint on an EAST-ADL:: Timing:: EventChain corresponding to the reception of inputs and production of outputs of the function.	MARTE:: Time:: TimedConstraint between MARTE:: Time:: TimedInstantObservations corresponding to the reception of inputs and production of outputs of the function.
Function connector communication time budget	No specific constraint for communication time budget. The execution time constraint can be used as a workaround.	MARTE:: Time:: TimedConstraint with interpretation = Duration between MARTE:: Time:: TimedInstantObservations corresponding to the start and the end of a communication.
End-to-end time budget	EAST-ADL:: Timing:: TimingConstraints:: ExecutionTimeConstraint on an EAST-ADL:: Timing:: EventChain corresponding to the end-to-end flow.	MARTE:: Time:: TimedConstraint between MARTE:: Time:: TimedInstantObservations corresponding to the end-to-end flow or use MARTE:: SAM:: SaEndToEndFlow
Period	EAST-ADL:: Timing:: TimingConstraints:: PeriodicConstraint	MARTE_Library:: BasicNFP_Types:: ArrivalPattern:: Periodic
Frequency	EAST-ADL:: Timing:: TimingConstraints:: PeriodicConstraint	MARTE_Library:: BasicNFP_Types:: ArrivalPattern:: Periodic
Inter arrival time	EAST-ADL:: Timing:: TimingConstraints:: SporadicConstraint	MARTE_Library:: BasicNFP_Types:: ArrivalPattern:: Sporadic

Table 4.1: Functional Modeling Languages

4.1.5 Design at the Functional Level

The functional level can be used to take some early design decisions. These decisions can for example include:

- separation of functions (allocate functions on different hardware components) because of safety constraints,
- optimization of timing or performance of the overall system by grouping functions that have more communication with each other,
- exploring the best hardware architectures to support the functional architecture.

This early design exploration requires to have at the functional level:

- modeling elements abstracting hardware platforms alternatives in terms of execution nodes and networks,
- modeling elements to capture allocation alternatives of FunctionComponents to nodes as well as FunctionConnectors to networks.

Moreover this early design activities at the functional level can be used to have:

- a more precise timing budgets estimation for functions and communication (because these elements are hardware dependent, so considering allocation of functions to hardware nodes let the designer specify more precisely functions timing budgets),
- some early timing validation at the functional level is possible (at least to discard unfeasible designs due to allocation scenarios leading to overloaded configurations),
- comparison of different functions to hardware allocation scenarios to guide the designer choice.

These early design decisions are further refined on the lower Autosar implementation levels.

4.2 Overview of Function-level Use Cases

This chapter describes timing-related use cases for system function analysis and design on functional level. Some use cases are covering the high-level timing in an early stage of the development while others are dealing with the transition from the functional level to the implementation level.
The design of a functionality requires its decomposition into function blocks. This decomposition must include an activity of decomposition of its timing requirements ending with an assignment of coherent timing requirements to function blocks.
Timing and load requirements are further split up when the function blocks are assigned to actual hardware and the communication technology is chosen. At this level all hardware is still abstract and detailed analysis considering e.g. priorities and scheduling are not yet possible. On the other hand it is possible to assign functions to abstract hardware resources and perform first analyses for load and latency. This allows to explore feasible partitioning solutions to be more confident with the timing requirements. Finally the requirements have to be considered when runnables/triggers are designed on the implementation level and verified when the hardware specification is known.

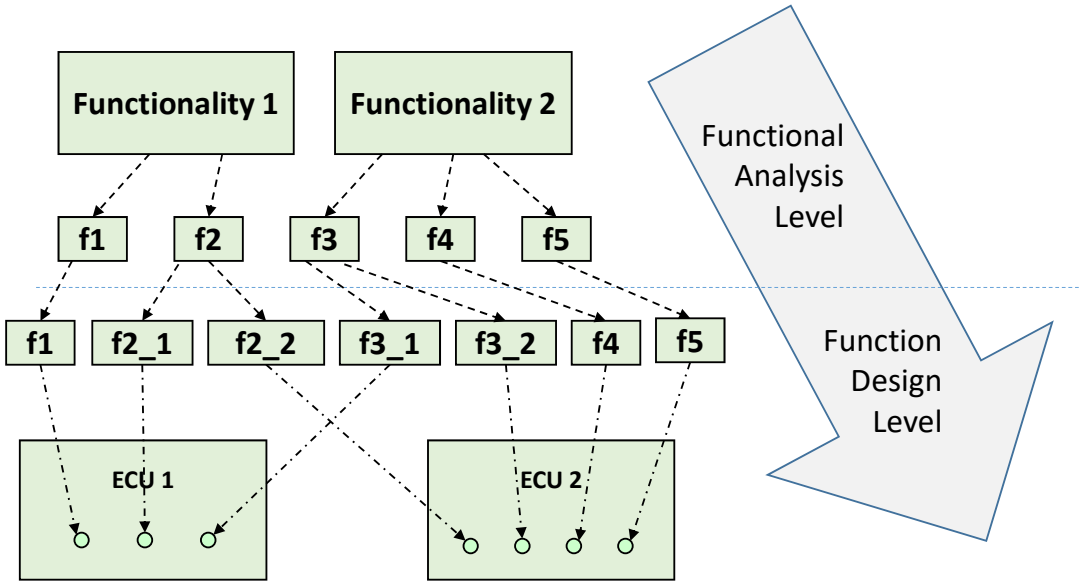

Figure 4.3: Decomposition of functions

The main goal during modeling and decomposition of functions from a timing perspective is to define the timing requirements on a functional analysis level and to refine and meet the requirements on the design and implementation levels.

The following use cases partly refer to the active steering example from chapter 1.4, see also figure 1.2 for an example overview and figure 1.3 for the Software Architecture.

Relation to other chapters Chapter 3 describes the decomposition of Timing Requirements in more detail. Chapter 5, 6 and 7 contain use cases for E2E, Network and ECU use cases. Chapter 9 contains timing properties and methods of interest for all use cases.
Links to explanations of the used timing expressions

- Load, see section 9.4
- Functional Analysis Level and Function Design Level, see section 3.3.1

List of use cases:

Section	Use case	Page
4.3	Function-level use case "Identify timing requirements for a new vehicle function"	58
4.4	Function-level use case "Partition a vehicle function into a Functional Architecture"	60
4.5	Function-level use case "Map a Functional Architecture to a hardware components network"	61

Section	Use case	Page

Table 4.2: List of Function-level specific use cases

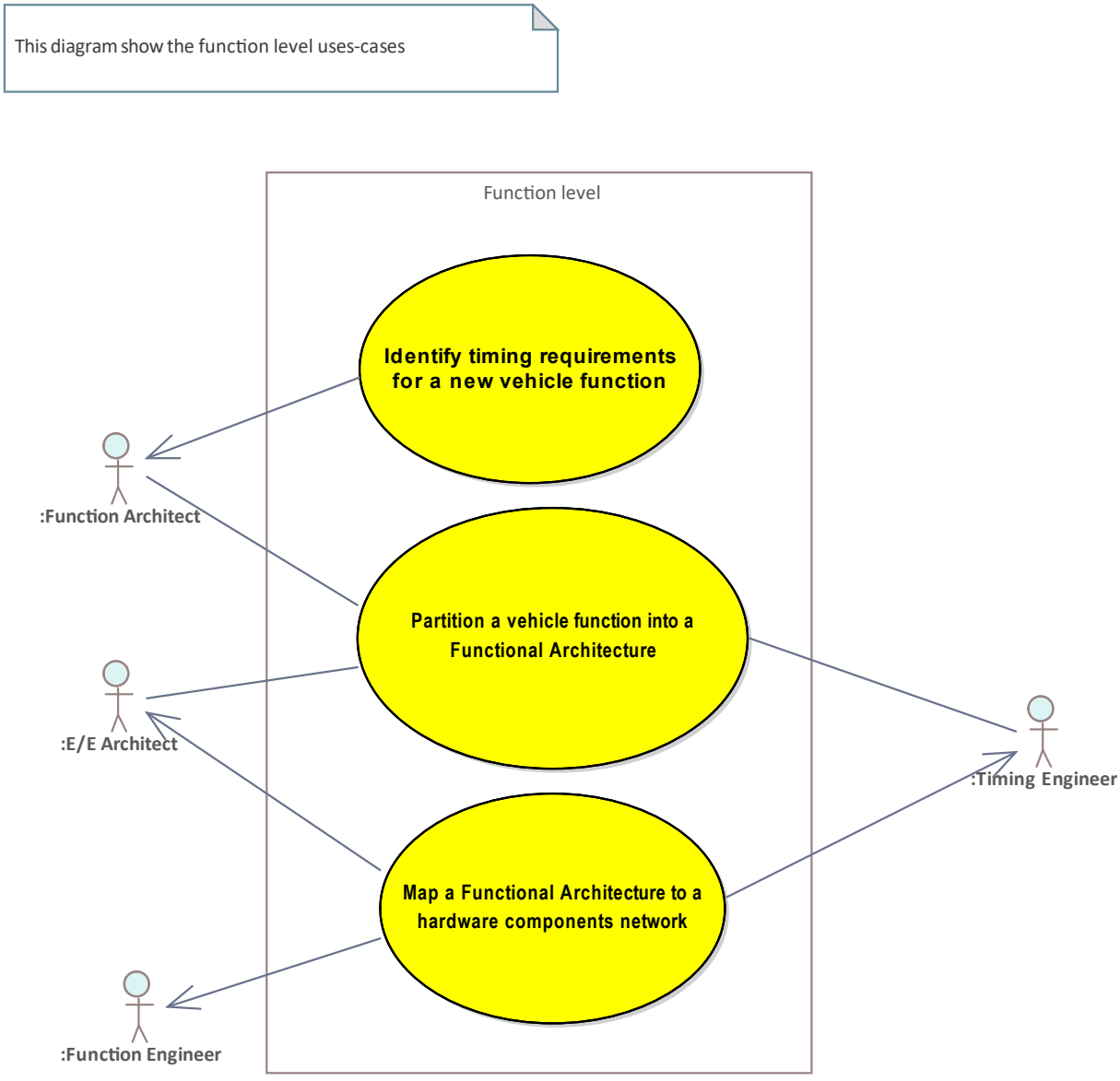

Figure 4.4: Use case Diagram: Function-level

4.3 Function-level use case "Identify timing requirements for a new vehicle function"

In the following use case, a vehicle function, is introduced to an existing functional architecture.

In this use case, a vehicle function is considered as black box and is not decomposed. Timing requirements on this level are typically derived from customer requiments, legal requirements, physics, etc. An example in the timing reference platform is the identification of a 150ms end-to-end requirement. (see "Identification of Timing Requirements" in Figure A.3 “Design of TRP on Functional Level”)

Goal In Context:	Identify timing requirements related to a new vehicle function.
Brief Description:	A new vehicle function is introduced. The objective of this use case is to identify timing requirements of the new vehicle function. The purpose of timing requirements is to be able to verify that the timing of the vehicle function fulfills its functional needs. For this the vehicle function is investigated thoroughly to identify timing critical event chains and establish constraints/bounds on acceptable timing. Timing requirements could be for example the maximum tolerable delay from changes in the sensor values to changed stimulus to actors. In example 1.4 the vehicle function "‘active steering"’ would come with a timing constraint that "‘the maximum delay between changes in the yaw rate sensor until electric motor stimulus is changed must be below 30ms"’ Ideally these timing constraints are formulated in a formal fashion (like in Definition and Classification of Timing Properties), they should refer to observable events as precisely as possible, and they should be independent of any actual implementation (i.e. do not refer to specific runnables or frames).
Scope:	Functional Architecture - Functional Analysis Level
Frequency:	During function development
Precondition:	The vehicle function is sufficiently specified to allow reasoning about acceptable timing, ideally through experiments or meaningful modeling or functional simulation.
Success End Condition:	All timing requirements related to this vehicle function are known.
Failed End Condition:	Some timing requirements could not be established, therefore not being testable later, opening the risk of integration problems.
Actor(s):	“Function Architect”

Table 4.3: Characteristic Information of "Identify timing requirements for a new vehicle function" use case

4.3.1 Main Scenario

A systematic approach for this use case is depicted in figure 4.5. The following steps typically apply:

1. The Function Architect verifies the description of the new vehicle function to ensure that all relevant details of the vehicle function are described and that there are no open questions.
2. The Function Architect performs preliminary analysis of the vehicle function. He then investigates the vehicle function with regards to user experience, technical limitations and safety goals or regulations. Based on the results of the investigation the Function Architect formulates timing requirements in the form of timing constraints(e.g. duration, latency, data age, period, jitter, ...).

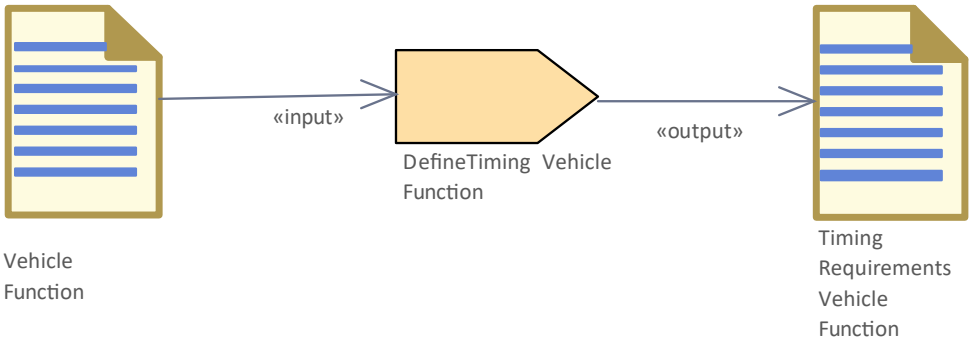

Figure 4.5: SPEM process model for Function-level use case "Identify timing requirements for a new vehicle function"

4.4 Function-level use case "Partition a vehicle function into a Functional Architecture"

In the following use case the vehicle function is refined and represented as a Functional Architecture which is a set of function blocks and their interfaces. The Functional Architecture represents all parts of the vehicle function that need to be executed on the vehicle's E/E-platform. Timing requirements associated with the vehicle function need to be decomposed as well and associated to the parts of the Functional Architecture. An example in the timing reference platform is the functional decomposition. (see "Partition into Functional Architecture" in Figure A.3 "Design of TRP on Functional Level")

Goal In Context:	Refine a vehicle function into a Functional Architecture with timing requirements.
Brief Description:	A vehicle function is usually a rather high-level specification of the intended behavior. In order to facilitate an efficient work on the following work steps, a more formal specification is required. For this the vehicle function is partitioned into function blocks and their interfaces (i.e. later implemented as intra- or inter-ECU communication). This is called the Functional Architecture. The timing requirements identified in Use Case 4.3 are associated with the function blocks and interfaces wherever possible.
Scope:	Functional Architecture - Functional Analysis Level
Frequency:	When integrating a new vehicle function or re-designing an existing vehicle function.
Precondition:	The vehicle function and its timing requirements are fully described. The principal logic of the vehicle function is known.
Success End Condition:	Vehicle function is successfully decomposed in function blocks. Interfaces between function blocks are defined and consistent. Timing requirements are associated with the function blocks and interfaces.
Failed End Condition:	Vehicle function cannot be decomposed in function blocks consistently.
Actor(s):	Timing Engineer, Function Architect, E/E Architect

Table 4.4: Characteristic Information of "Partition a vehicle function into a Functional Architecture" use case

4.4.1 Main Scenario

A systematic approach for this use case is depicted in figure 4.6. The following steps typically apply:

1. The E/E Architect analyses the description and requirements of the vehicle function and formalizes it, by describing function blocks and there interactions through interfaces.
2. The Function Architect takes the timing requirements specified for the vehicle function and associates these requirements to the function blocks and interfaces from the decomposition of the vechicle funtion.
3. The Timing Engineer verifies the timing requirements of the vehicle function against the timing requirements of the Functional Architecture(e.g. duration, latency, data age, period, jitter, ...), to ensure consistency between those requirements. E.g. the overall latency of the vehicle function has to be consistent with the latency, period, jitter, ... of the decomposed function blocks.

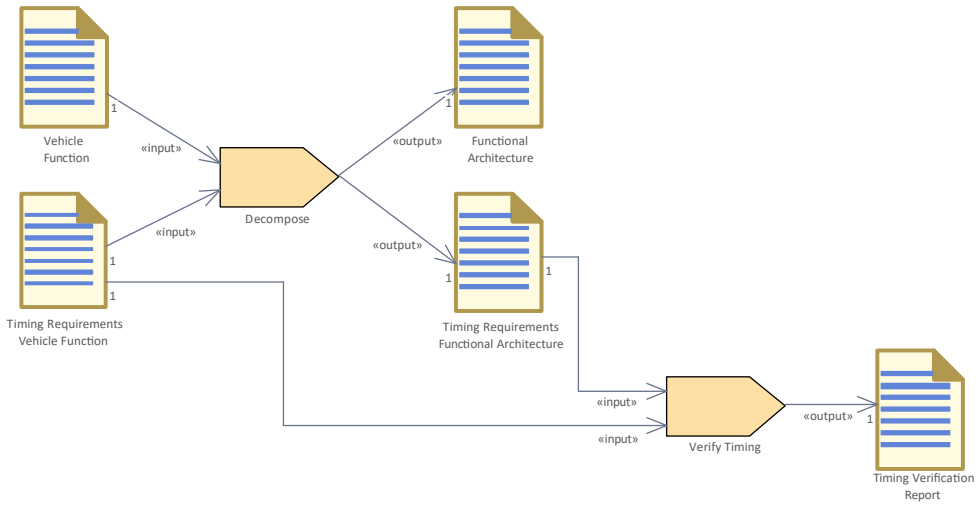

Figure 4.6: SPEM process model for Function-level use case "Partition a vehicle function into a Functional Architecture"

4.5 Function-level use case "Map a Functional Architecture to a hardware components network"

An example in the timing reference platform is the mapping to virtual hardware. (see "Map to Virtual Hardware" in Figure A.3 “Design of TRP on Functional Level”)

In the following use case an abstraction of the vehicle’s E/E architecture in the form of a network of hardware components is added to the functional architecture model.

Goal In Context:	Specify a mapping of a Functional Architecture to a virtual hardware components network

Brief Description:	This use case consists in finding a mapping of the Functional Architecture to the hardware network, and the goal is that this mapping is compliant with timing requirements of the vehicle function. This allows CPU load balancing and decissions for suppliers in an early stage. Function blocks that already exist as part of other functions, or whose interfaces are reused shall be checked for consistency and may need to be revised. A meaningful constraint is to assume that a function block is mapped to exactly one hardware component.
Scope:	Functional Architecture - Functional Design Level
Frequency:	During function partitioning
Precondition:	A Functional Architecture with timing requirements for each function is specified. A hardware network is available (ideally in the form of a model with hardware component's main characteristics).
Success End Condition:	Each function block and its interfaces are mapped to hardware components. The early evaluation of the mapping satisfies timing and load requirements.
Failed End Condition:	Some functions could not be mapped to hardware components, or the mapping evaluation does not satisfy timing and load requirements, opening the risk of overload problems.
Actor(s):	Timing Engineer, Function Engineer, E/E Architect

Table 4.5: Characteristic Information of "Map a Functional Architecture to a hardware components network" use case

4.5.1 Main Scenario

A systematic approach for this use case is depicted in figure 4.7. The following steps typically apply:

1. The E/E Architect checks that the functional architecture description is complete, all timing requirements for each function and function chain is specified and if available, that the E/E architecture model is consistent with the functional architecture.
2. With the assistance of the Function Engineer, the E/E Architect tries to find suitable virtual hardware components to map the function blocks on to. It needs to be ensured that all required hardware resources for a function block are available and that the interfaces between the function blocks can be connected, while also satisfying the timing and load constraints.
3. The Timing Engineer verifies the function to ECU mapping against the timing requirements from the functional architecture timing description(e.g. budgets, period, jitter, ...) and documents the results in the timing verification report.
4. If the timing verification report reveals any deficiencies, the E/E Architect needs to find another mapping solution that resolves the timing violations. In case a feasible mapping cannot be found, it may be required to upgrade the virtual hardware components network to meet the timing requirements.

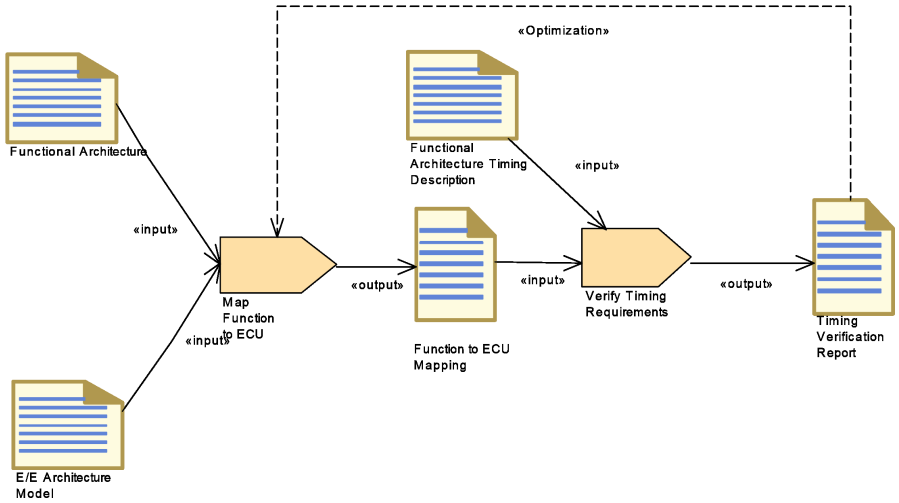

Figure 4.7: SPEM process model for Function-level use case "Map a Functional Architecture to a hardware components network"

5 End-to-End Timing for Distributed Functions

This chapter introduces use cases to reason about the end-to-end timing of distributed functions. As a distributed function, we consider

- a function that executes locally but requires data from sensors or functions communicated over the network. In this case there exists at least an assumption about the maximum age of the data or
- a function that consists of several computation steps that are performed on different ECUs, connected via dedicated or shared buses. In this case event chains often exist with overall latency or periodicity constraints.

Most automotive functions today are distributed functions.

5.1 Relation to other chapters

This chapter is related to the other parts of this document as follows:

- Chapter 3 introduces the terminology of a function, what a function is, and informally discusses how an end-to-end timing requirement can be decomposed into timing requirements for each involved resource. Further, use cases on function level are treated in this chapter.
- Chapter 4 contains use cases on function level.
- Chapter 6 and 7 discuss use cases related to timing analysis on individual resources.
- Chapter 9 defines timing properties and how to derive these for individual resources and schedulables. The relevant timing properties (in particular the load of a resource and the latency of a schedulable) are introduced in Definition and Classification of Timing Properties.

Furthermore, the chapter (and the other chapters) is related to the AUTOSAR Timing Extensions [2], as it allows to derive the guarantees or assert the constraints specified therein.

5.2 Overview of End-to-End Use Cases

This chapter describes the following use cases listed in Table 5.1.

Section	Use-case	Page
5.3	E2E use case "Derive per-hop time budgets from End-to-End timing requirements"	66
5.4	E2E use case "Deriving timing requirements from an existing implementation"	68

Section	Use-case	Page
5.5	E2E use case "Specify Timing Requirements for functional interfaces based on Signals/Parameters"	69
5.7	E2E use case "Verify guarantees against timing requirements"	72
5.8	E2E use case "Trace-based timing verification of a distributed implementation"	74
5.9	E2E use-case "Introduction of Service-Oriented Communication"	76

Table 5.1: List of use cases related to end-to-end timing

Figure 5.1: Use case diagram: E2E

5.3 E2E use case "Derive per-hop time budgets from End-to-End timing requirements"

This use case specifies the work flow for function(s) owners on how to decompose end-to-end timing requirements in order to derive and specify per hop time budgets (e.g. define local time budgets for functions(s) for each execution node and communication bus they are distributed).

This use case becomes relevant when a customer functionality is identified in the specification. This functionality is decomposed into a set of functions that have to be integrated into an existing E/E network. An end-to-end timing requirement is identified for these functions and time budgets for each segment of the end-to-end chain have to be derived. A suitable syntax to store this and related properties is provided by the AUTOSAR Timing Extensions [2].

Goal In Context:	Specify time budgets for each ECU and communication network on which function(s) participating in an end-to-end timing requirement are distributed.
Brief Description:	This use case requires that an end-to-end timing chain of a function or a set of functions with an end-to-end timing requirement is specified. Moreover, functions participating in the end-to-end timing chain are decomposed in sub-functions that are allocated to ECUs interconnected with communication networks. Based on end-to-end timing requirements, this use case derives time budgets for each sub-function (segment of the end-to-end timing chain) allocated to an ECU and each involved network segment.
Scope:	E2E
Frequency:	Whenever a new distributed function has to be implemented
Precondition:	An end-to-end timing chain (with a function decomposition) with an end-to-end timing requirement is available.
Success End Condition:	A time budget has been found for each sub-function (segment) of the end-to-end timing chain and the sum of the budgets are not exceeding the end-to-end timing requirement.
Failed End Condition:	Some time budgets could not be derived or the sum of the found time budgets exceeds the end-to-end timing requirement.
Actor(s):	Timing Engineer, Function Engineer, Network Data Engineer, ECU Integrator.

Table 5.2: Characteristic Information of E2E use case "Derive per-hop time budgets from End-to-End time requirements"

5.3.1 Main Scenario

A systematic approach for this use case is depicted in figure 5.2. The following steps typically apply:

1. A distributed functionality is identified and decomposed into a set of manageable functional components (functions). An end-to-end timing requirement is identified from the specification (or established from experiments). A suitable property to describe the end-to-end timing requirement is given by GENERIC PROPERTY Latency which can be determined in the context of Task “Collect Timing Requirements”.

2. The Function Engineer decomposes the function and distributes the sub-functions (functional contributions of components) on a network of ECUs. This results in an allocation of functions on computation and communication resources. In case an end-to-end timing requirement spans across different networks, segments for routing between networks should be included in the time budget planing.
3. Optionally, the ECU Integrator and the Network Data Engineer are asked for an estimation of the required processing time (estimated timing guarantees). A suitable method is GENERIC METHOD Determine Latency either via implementation-based timing analysis (Table 10.10) or model-based timing analysis (Table 10.9).
4. Time budgets for each sub-function are derived from the end-to-end timing requirement, if possible respecting the estimated per hop timing.
5. A verification of the found time budgets with respect to the end-to-end timing requirement is performed by the Timing Engineer.

See also related sub-use-case: NW use case “Integration of new communication”.

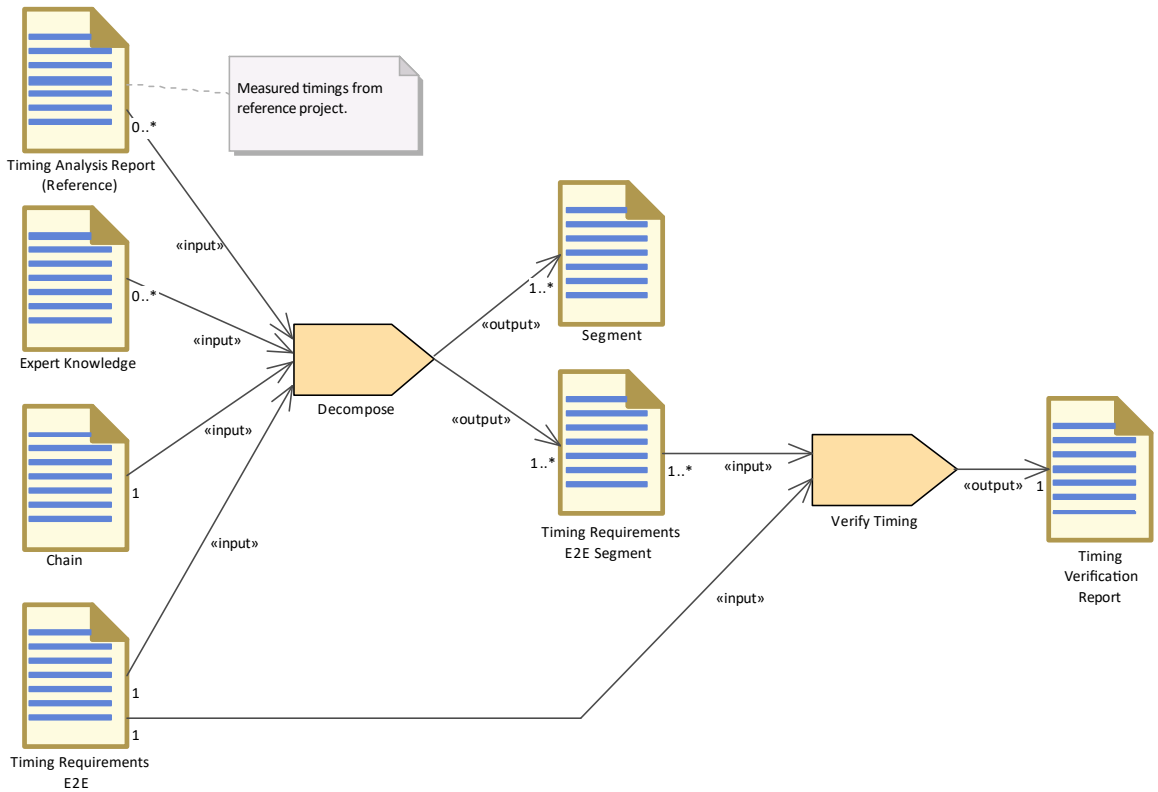

Figure 5.2: SPEM process model for E2E use case "Derive per-hop time budgets from End-to-End timing requirements"

5.4 E2E use case "Deriving timing requirements from an existing implementation"

This use case specifies the work flow for Function Engineers to derive timing requirements (e.g. deadlines, end-to-end timing) for a system under development from a previous implementation-based timing analysis of an existing implementation. This may be applied at early phases in the design process for a new build system where timing requirements can not be derived by the implementation-based/model-based timing analysis due to e.g. an incomplete system / missing functions, a not existing/specified hardware platform or a missing setup for timing analysis. Moreover this use case may be applied in case of the migration from a single-core platform to a multi-core platform to where as the timing requirements derived from an existing single-core implementation may be applied to the multi-core implementation of this system. Further possible application are the migration from one network type to another or replacement of an old hardware by a new one. The goal of this use case is that the timing behavior of the system under development sufficiently corresponds to the timing behavior of the existing implementation (i.e. timing is identical or better).

The AUTOSAR Timing Extensions (TIMEX) [2] represent a suited grammar to formalize the description of timing constraints.

Goal In Context:	Specification of timing requirements for systems under development based on a timing analysis of existing implementations (early in the design process without knowledge and access of the final and complete system implementation and behavior).
Brief Description:	This use case describes the deriving of timing requirements for a system under development from timing of an existing implementation of this system or parts of this system.
Scope:	E2E, NW, ECU
Frequency:	Whenever an existing function has to be implemented on a new system.
Precondition:	A timing analysis of an existing implementation of a function for the system under development is available or can be performed on a similar system.
Success End Condition:	The timing requirements derived from a timing analysis of an existing implementation are mapped and applied to a system under development
Failed End Condition:	The timing requirements derived from a timing analysis of an existing implementation could not be accessed, mapped and/or applied to a system under development
Actor(s):	Timing Engineer, Function Engineer, Test Engineer.

Table 5.3: Characteristic Information of this E2E use case

5.4.1 Main Scenario

A systematic approach for this use case is depicted in figure 5.3. The following steps typically apply:

1. If a timing analysis is not available, the Timing Engineer and Test Engineer need to perform a timing analysis of an existing implementation of the system. A suitable property to describe the end-to-end timing requirement is given by GENERIC

PROPERTY Latency, which can be determined using GENERIC METHOD Determine Latency (for timing analysis of the existing implementation) in the context of Task “Perform Implementation-Based Timing Analysis”. In case an end-to-end timing requirement spans across different networks, the timing property SPECIFIC PROPERTY Response Time (Routing) can be relevant and can be obtained using timing method SPECIFIC METHOD Determine Response Time (Routing).

2. The Function Engineer derives the timing requirement and maps/applies it to the system under development. See Task “Collect Timing Requirements”.

See also related sub-use-cases: NW use case “Remapping of an existing communication link”, ECU use case “Collect Timing Information of a SWE” and ECU use case “Create Timing Model of the entire ECU”.

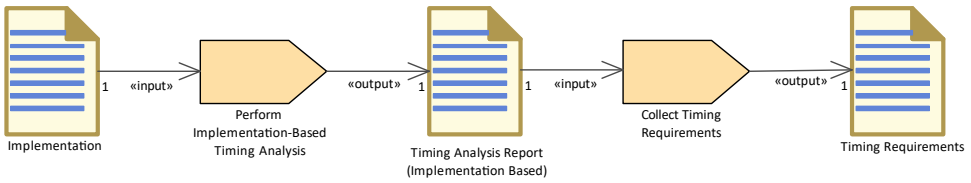

Figure 5.3: SPEM process model for E2E use case "Deriving timing requirements from an existing implementation"

5.5 E2E use case "Specify Timing Requirements for functional interfaces based on Signals/Parameters"

This use case specifies the work flow for Function Engineer (e.g. of a distributed function), to derive and specify the relevant timing-related properties and requirements in order to consider its communication in the vehicle networking.

Goal In Context:	To specify precisely the requirements of a function with respect to the required data communication over the vehicle network.
Brief Description:	This use case requires dedicated reasoning about the timing requirements of a specific function. The Function Engineer identifies for each signal/parameter (i.e. the data to be communicated over the network) the expected cycle time, jitter and latency. To ensure that the requirement is not over-specified the requirements are reviewed by the Network Data Engineer.
Scope:	E2E, NW
Frequency:	Whenever the communication requirements of a function changes
Precondition:	The end-to-end timing constraints for the involved signals into ECU-internal timing requirements and network-related timing requirements have been partitioned.(see also Use Case 6.3)
Success End Condition:	The function’s timing requirements have been considered in an explicit signal/-parameter request.
Failed End Condition:	The function’s timing requirements could not be translated to a signal/parameter request (e.g. because they are not known).
Actor(s):	Function Engineer, Network Data Engineer.

Table 5.4: Characteristic Information of this E2E use case

The signal/parameter request shall contain the following information

- the name of the signal/parameter
- the size of the signal/parameter
- the receivers of the signal/parameter as a list of ECUs and/or software components
- the maximum tolerated age of the signal when transmission is completed on the target network.
- the expected update frequency of the signal/parameter
- the accepted jitter for the transmission of the signal/parameter
- a short description of the related functionality

From these values, typically, a signal-to-frame (parameter-to-package) mapping will be derived. In case of designing a CAN configuration, the requesting function owner receives the following parameters for consideration in ECU development:

- the name of the frame
- the transmission property (e.g. periodic)
- the frames cycle time (if relevant)

This information then becomes part of the AUTOSAR System Description.

5.5.1 Main Scenario

This use case typically consists of the following steps:

1. The Function Engineer specifies a signal/parameter request that (if so implemented) enables a correct operation of the function.
2. The Network Data Engineer investigates the signal/parameter request and reviews its content for completeness and adequacy. Indications for non-adequate signal/parameter requests may be if the maximum tolerated age is smaller than the update frequency or if an update frequency of less than the period of the involved tasks is requested. In case of such irritations, the Network Data Engineer and the Function Engineer iterate until an adequate request has been identified. Related to this step are the methods listed in Table 9.40 and the properties listed in Table 9.12.
3. The Network Data Engineer documents and files the signal/parameter request.

4. (in the following, the Network Data Engineer will consider the signal/parameter request to find a suitable signal-to-frame (parameter-to-package) mapping and routing entry for the signal)

See also related sub-use-case: NW use case “Integration of new communication”

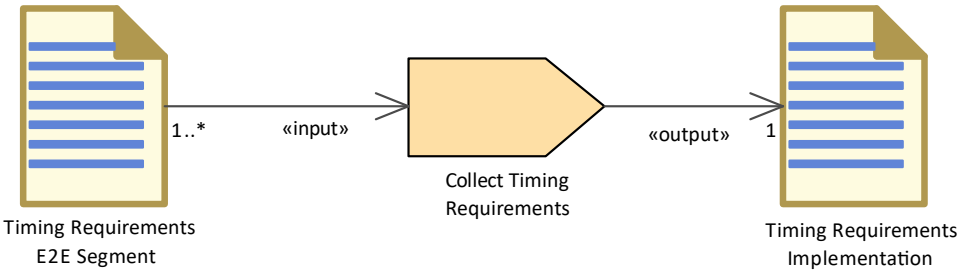

Figure 5.4: SPEM process model for E2E use case "Specify Timing Requirements for functional interfaces based on Signals/Parameters"

5.6 E2E use case "Aggregate E2E timing guarantees from per-hop timing guarantees"

This use case aggregates latency property values derived by analysis of the actual implementation ("guarantees") of a segment to timing guarantees of an end-to-end timing chain.

For the verification of an end-to-end timing requirement of a distributed function it may be required to aggregate the latency property values measured for individual segments of the end-to-end timing chain. This can be due to an early stage of development where only individual parts of the system exist or that the timing analysis is only possible for individual parts, because of technical limitations.

In the trivial case the aggregation can be as easy as the sum of the latency property values of the segments. But latency property values are usually influenced by the operating conditions and can have varying impacts on each segment. If for example you are interested in the worst-case latency of an end-to-end timing chain, the worst-case latency of two segments of this chain may never occur under the same operating conditions. So the actual worst-case latency can be lower then the sum of the worst-case latencies of the two segments. If the latency property values are represented by a probability distribution, there are analytical methods or a Monte-Carlo simulation based approaches to combine the probability distribution of two segments.

Goal In Context:	Aggregate a latency property values for an end-to-end timing chain from the timing properties of each of its segments.
Brief Description:	This use case is applicable if timing properties derived by analysis of the actual implementation of segments need to be aggregated to determine latency property values for an end-to-end timing chain.
Scope:	System
Frequency:	Whenever timing guarantees change.

Precondition:	The timing guarantees for all segments of an end-to-end timing chain are known. The timing guarantees are available for all relevant operating conditions. There is no gap or overlap, between neighboring segments.
Success End Condition:	A guarantee for the end-to-end timing chain is determined.
Failed End Condition:	-
Actor(s):	Timing Engineer

Table 5.5: Characteristic Information of this E2E use case

5.6.1 Main Scenario

A systematic approach for this use case is depicted in figure 5.5. The following steps typically apply:

1. The Timing Engineer checks, that Timing Analysis Reports (see table 10.12) for all segments are available and cover all relevant operating conditions.
2. The Timing Engineer extracts the latency property values for the same operating conditions from the Timing Analysis Reports of each segment. Segment by segment the latency property values are aggregated until the timing properties of the complete chain have been determined. The Timing Engineer has to pay attention that the end point and start point of two neighboring segments match. E.g. for GENERIC PROPERTY Latency, if the segments overlap or there is a gap between the segments the aggregated value will be too long or too short.
3. The Timing Engineer composes a new Timing Analysis Report for the end-to-end timing chain with the aggregated latency property values.

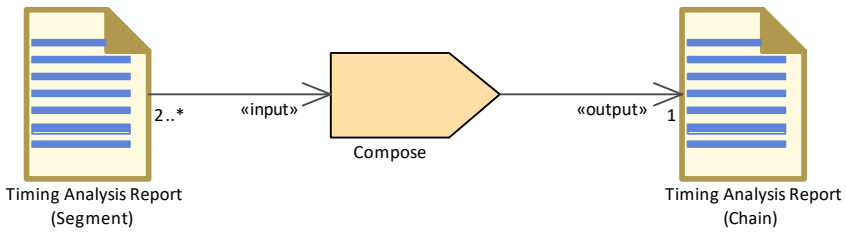

Figure 5.5: SPEM process model for E2E use case "Aggregate E2E timing guarantees from per-hop timing guarantees"

5.7 E2E use case "Verify guarantees against timing requirements"

This use case compares the timing properties derived by analysis of the actual implementation ("guarantees") to the system's specification ("requirements").

Since the development of the different parts of an E2E chain is split across different departments and organizations collecting and verifying guarantees regularly helps to ensure that the timing requirements can be achieved and issues can be corrected early on. This should be started as early as possible during development. First guarantees

can be acquired from estimates for the initial implementation of a component, getting reference values from an existing project, measurements of reused software on the previous hardware or performing simulations once a first configuration for a communication link is known. As the development progresses the accuracy of the guarantees improves as the timing-models mature with every iteration and the measured software and hardware gets closer to the final product.

The best outcome of this use case is if the requirements are fulfilled by the guarantees. Otherwise, either requirements need to be relaxed or the guarantees have to be improved (e.g. by reconfiguration of the system).

Goal In Context:	Verify whether the timing of a specific implementation adheres to the timing requirements.
Brief Description:	This use case establishes the comparison of the analysis results of the actual implementation ("guarantees") to the intended behavior as specified ("requirements").
Scope:	ECU, NW, E2E
Frequency:	Whenever timing requirements or timing guarantees change.
Precondition:	All relevant timing requirements and timing guarantees must be known, any timing requirement that has not been quantified and listed specifically will not be considered in the evaluation.
Success End Condition:	It is known whether all timing requirements are fulfilled by the current implementation. Best outcome is if the requirements are fulfilled by the guarantees. Otherwise, either requirements need to be relaxed or the guarantees have to be improved.
Failed End Condition:	At least one guarantee causes a violation of a timing requirement.
Actor(s):	Timing Engineer, Test Engineer.

Table 5.6: Characteristic Information of this E2E use case

5.7.1 Main Scenario

A systematic approach for this use case is depicted in figure 5.6. The following steps typically apply:

1. Depending on the progress of the development, guarantees will only be available for segments of an E2E chain. In this case the individual guarantees can be asserted against the timing requirements from E2E use case "Derive per-hop time budgets from End-to-End timing requirements" Once the guarantees can be acquired end-to-end, they can be directly verified against the E2E timing requirement.
2. Establish best known guarantees provided by evaluation of the implementation by performing Task “Perform Model-Based Timing Analysis” or Task “Perform Implementation-Based Timing Analysis”. Related to this steps are the methods listed in Table 9.40, the generic methods described in Section 9.5 and the properties listed in Table 9.12.

3. The guarantees of the individual segments have to be aggregated as described by E2E use case "Aggregate E2E timing guarantees from per-hop timing guarantees".
4. The guarantees, requirements and the comparison of the two are reported (Task “Verify Timing”).

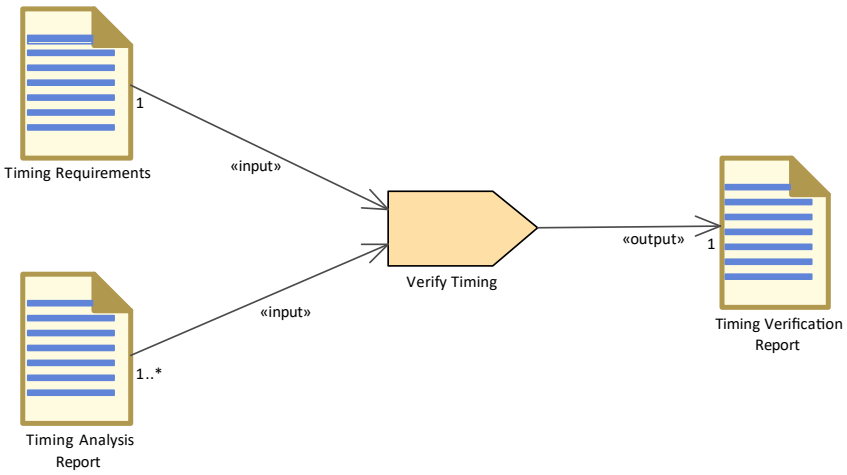

Figure 5.6: SPEM process model for E2E use case "Verify guarantees against timing requirements"

5.8 E2E use case "Trace-based timing verification of a distributed implementation"

Whether for understanding, debugging or verifying the timing behavior of a distributed system, tracing of the relevant buses and ECUs (hereafter referred to as subsystems) significantly simplifies timing analysis.

This is even more true if the traces from the various subsystems can be aligned (i.e. synchronized) in order to show cross-subsystem timing effects of event chains such as cross-core communication in a multi-core system, data-buffering-effects when an ECU send/receives data to/from a communication network or even complete end-to-end timing scenarios.

Goal In Context:	Understand, debug and verify the timing behavior of a distributed implementation.
Brief Description:	Tracing observes the real system. For dedicated events such as a start of a task or the presence of a certain message on a bus, timestamps together with event information is placed in a trace buffer which can later be used to reconstruct and analyze the observed scenario. For details, see Measurement and Tracing. For analyzing cross-subsystem timing effects, it becomes necessary to synchronize the traces from all of the relevant subsystems.
Scope:	E2E, ECU, NW
Frequency:	Whenever timing information about the actual implementation are needed

Precondition:	Existing and executable system, accessible subsystems. Tracing tools have to be in place, licensed and integrated into the system for the test engineer to use.
Success End Condition:	Tracing performed and data (=traces) ready for analysis; if necessary, traces from different subsystems (cores, ECUs, buses) are aligned, i.e. synchronized.
Failed End Condition:	No or not all relevant scheduling entities could be traced or traces could not be aligned (i.e. synchronized)
Actor(s):	Timing Engineer, Test Engineer.

Table 5.7: Characteristic Information of this E2E use case

5.8.1 Main Scenario

A systematic approach for this use case is depicted in figure 5.7. The following steps typically apply:

1. The Timing Engineer and Test Engineer prepares the measurement and the system under test (tools, software...).
2. The Test Engineer performs a correlated (i.e. synchronized) tracing of an existing implementation of the system under consideration. See Task “Perform Implementation-Based Timing Analysis”. Related methods and properties are GENERIC METHOD Determine Latency, GENERIC METHOD Determine Load, GENERIC PROPERTY Latency and GENERIC PROPERTY Load. In case an end-to-end timing requirement spans across different networks, the timing property SPECIFIC PROPERTY Response Time (Routing) can be relevant and can be obtained using timing method SPECIFIC METHOD Determine Response Time (Routing).
3. The Timing Engineer checks the quality of the traces and if quality is sufficient. Add time compensation to traces for synchronization if required. Compose Timing Analysis Report from the collected trace data.
4. With the Timing Analysis Report it is possible for the Timing Engineer to verify the timing behavior against the Timing Requirements.

See also related sub-use-cases: ECU use case “Collect Timing Information of a SWE”, ECU use case “Collect Timing Information of a SWE” and ECU use case “Verification of Timing”.

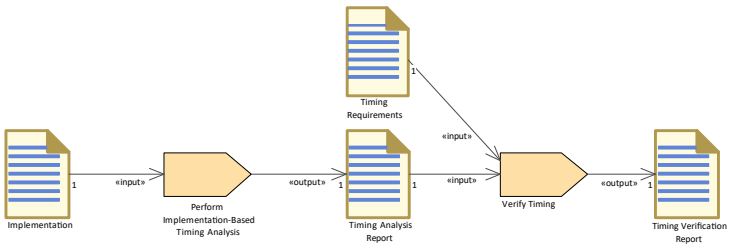

Figure 5.7: SPEM process model for E2E use case "Trace-based timing verification of a distributed implementation"

5.9 E2E use-case "Introduction of Service-Oriented Communication"

Distributed systems based on Classic Platform use sender/receiver or client/server communication between SWCs. On buses signal based communication is used in many cases, but with SOME/IP Transformer first steps towards service-oriented communication are possible. The Adaptive Platform is completely based on service-oriented communication. When new ECUs based on Adaptive Platform are introduced into the distributed system, service-oriented communication has to be integrated and considered in E2E timing analysis.

This use case covers two aspects that are relevant for E2E timing: On the one hand service discovery is relevant for initial delays. One the other hand the kind of communication has strong impact on E2E latency. The general impact of these two aspects is discussed in this E2E use case, while details of the underlying network protocol (SOME/IP, DDS, ...) will be discussed in a network use case.

Service-oriented communication allows adaptive changes in the communication matrix: A service provider can start and stop publishing the services at any point in time and the subscriber can subscribe and unsubscribe at any time. To perform a subscription to a service the service has to be found. This process is called service discovery. The service discovery can be performed at start up of the system, at start of an application or when a function needs the service. Details of the service discovery process are implementation specific: e.g. a service can be announced periodically or just by adding it to a central registry. In the latter case the time that is needed for service discovery has to be considered for E2E latency of a distributed functions. If service discovery is performed at startup or if the services are statically configured at design time, service discovery is not part of E2E latency. Each client that subscribes to a service introduces additional load on the service provider. This also impacts the E2E latency.

The kind of communication is the most relevant factor in E2E latency of service-oriented communication: Events that are sent periodically by the service provider will arrive at the subscriber at a point in time that is hard to predict exactly due to high jitter introduced by stack and network. Especially in case of periodic processing of the received data this can introduce additional latencies. If method calls are used instead

of events, the execution of the method call is hard to predict from the servers point of view, that can also cause longer latencies.

Goal In Context:	Model based design, analysis and verification E2E timing of service-oriented communication.
Brief Description:	Define timing relevant parameters for service-oriented communication.
Scope:	ECU, NW, E2E
Frequency:	Whenever service-oriented communication is designed, modified or has to be verified.
Precondition:	Interfaces of servers and clients have to be specified. Model of remaining system, if available.
Success End Condition:	Specification of service-oriented communication that fulfills the requirements.
Failed End Condition:	Specification of service-oriented communication that does not fulfills at least one requirement.
Actor(s):	Timing Engineer, E/E Architect, Function Engineer.

Table 5.8: Characteristic Information of this E2E use case

5.9.1 Main Scenario

A systematic approach for this use case is depicted in figure 5.8. A common way to analyze the impact of introduction of service-oriented communication on E2E latencies is model based timing analysis. The following steps typically apply:

1. The Function Engineer collects information between which components service-oriented communication shall be introduced
2. The Timing Engineer creates a timing model of the system which covers the planned service discovery.
3. The Timing Engineer performs model based timing analysis with focus on service discovery.
4. The Timing Engineer creates a timing model of the system which covers the planned kinds of communication.
5. The Timing Engineer performs model based timing analysis with focus on kinds of communication.
6. The Timing Engineer verifies the result of the timing analysis against the timing requirements.
7. E/E Architect and Function Engineer optimize the service-oriented communication until all E2E requirements are fulfilled.

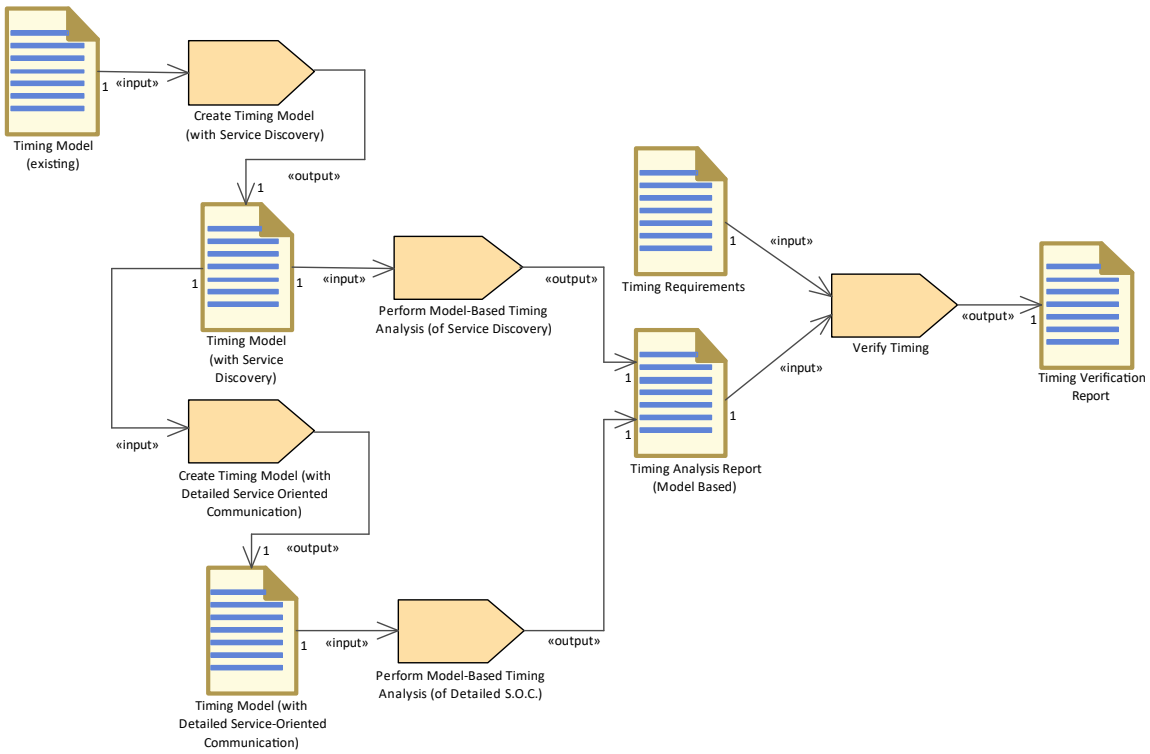

Figure 5.8: SPEM process model for E2E use-case "Introduction of Service-Oriented Communication (SOC)"

5.9.2 Validation in Timing Reference Platform

The use case "Introduction of Service-Oriented Communication" is the first one that is focused on Adaptive Platform. For Adaptive Platform a demonstrator is available that is the base for the Timing Reference Platform (TRP). It will be used for validation of this use case and to gather experiences for new use cases. For this first step, a system of two ECUs based on Adaptive Platform is required. One ECU runs an application that provides a service, the other ECU runs an Application that subscribes to the service. The communication shall consist of events and method calls. With this setup the latencies for service discovery and different communication types can be analyzed. For details on Timing Reference Platform see Appendix A.

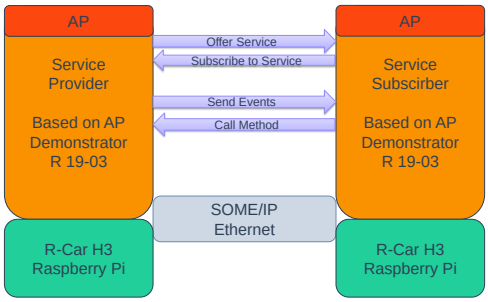

Figure 5.9: E2E use-case "Introduction of Service-Oriented Communication" in Timing Reference Platform

6 Timing for Networks

This chapter outlines timing use cases related to automotive network communication. The ECU related timing aspects covered by Chapter 7 may have direct or indirect impact on the network timing.

In an automotive communication network the timing behavior is mainly described by the communication matrix which contains the communication frames/package with the protocol and timing specific parameters (e.g. payload, IDs, frame triggering parameters).

Depending on the amount of traffic to be transmitted on the network and the communication protocol, a network may or may not satisfy given timing requirements such as maximum latency of frames or a given bus load threshold. In general, the network architect must define the parameters such that the timing requirements are fulfilled.

The use cases described in this chapter present problems and solutions related to the design of communication networks. Typical terms used in this chapter are:

- Load, see section 9.4
- Latency, see section 9.4
- Response Time, see section 9.4

6.1 Example

In Chapter 1, an example of a system with end-to-end timing requirements was given, see figure 1.2 on page 16. On the network, the end-to-end timing requirements are influenced by the network configuration and the scheduling of the network components, mainly buses and gateways, see figure 6.1. The use cases in this chapter refer to this example.

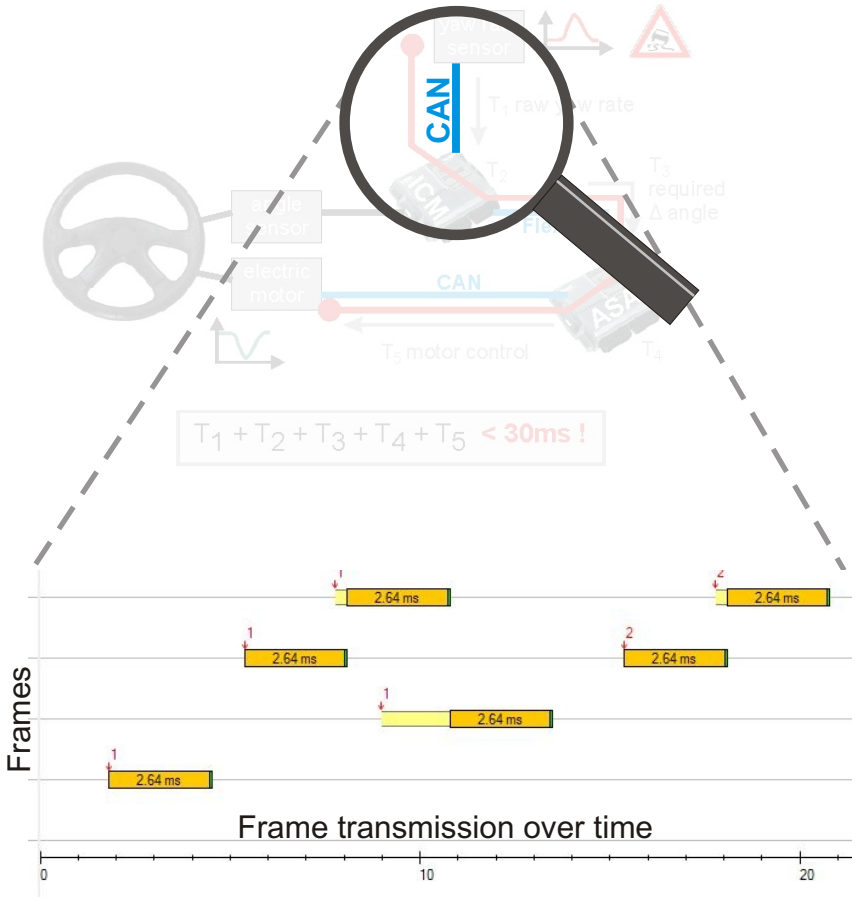

Figure 6.1: Focus of this chapter: bus timing in networks

6.2 Overview of Network Use-cases

The network use cases cover four cases:

1. Introducing new communication into an existing network (which already has a stable topology)
2. Establishing a new communication network (where the topology is not yet defined)
3. Remapping of an existing communication link of an existing network
4. Optimizing the timing properties of a communication network

The use cases require understanding the timing properties as a first step. A common example for such a property is the response time of a network message. Properties are defined as either a constraint or requirement as described in chapter 9.2.0.4.

Based on the identified timing properties, the timing requirements of the existing (or to be designed) network have to be collected and assessed.

Next step is to use these properties in order to build or extend a timing model. The timing model is required in order to verify the timing properties before start of implementation and to compare the implementation against the specification. Collected results from this step form the timing verification report.

This process can and should be executed iteratively to advance from verification to validation.

The detailed use cases are listed in Table 6.1.

Section	Use Case	Page
6.3	NW use case “Integration of new communication”	83
6.4	NW use case “Design and configuration of a new network”	87
6.5	NW use case “Remapping of an existing communication link”	90
6.6	NW use case “Changes of the E/E-Topology”	93
6.7	NW use case “Optimizing the communication timings”	97
6.8	NW use case “Derive timing properties of a message on a network segment”	100

Table 6.1: List of network specific use cases

Figure 6.2: Use case Diagram: Timing Analysis for Network

6.3 NW use case “Integration of new communication”

This use case focuses on integrating new communication into an existing automotive network.

Goal In Context:	Feasible integration of new communication into an existing networked architecture.

Brief Description:	Considering an E/E automotive architecture consisting of several ECUs connected via buses, it is required to integrate additional communication into the network, such that the legacy communication and the new communication fulfil the timing requirements. For example, the new communication is additional sensor data transmitted over the network from the *yaw rate sensor* to the *ASA*-ECU, as shown in figure 1.2 on page 16. The maximum latency of the new sensor data on the network bus must not exceed 10ms. The buses on which the new communication must be integrated may implement different communication protocols (e.g. CAN, LIN, Flexray, etc.). The communication on each bus is specified by a communication matrix containing the PDUs/frames with their protocol specific parameters and the communication behavior (timing parameters).
Scope:	System
Frequency:	Regular
Precondition:	For the new communication following properties are defined: • The size of the communication signals (SW-C Template / GenericStructureTemplate). • The transmitter and receiver nodes / system mapping • The PDU/`Frame` timing/triggering • Required bandwidth Additionally, for the communication on the network a set of timing requirements is known: • Maximum bus load on each bus • Maximum latency (e.g. response times, routing times) for each PDU/`Frame` Furthermore, a specification of the communication paradigm for the existing bus controllers is available, e.g. the CAN controller sends CAN-frames with different identifiers via a queue (priority ordered or FIFO), while different instances of the same frame are send via a register (always send the newest frame instance). It is assumed that the current network configuration satisfies the timing requirements.
Success End Condition:	The new communication was completely defined and the timing requirements are satisfied.
Failed End Condition:	The new communication cannot be defined without violating at least one timing requirement.
Actor(s):	Timing Engineer, Network Data Engineer

Table 6.2: Characteristic Information of NW UC “Integration of new communication”

6.3.1 Main Scenario

For the sake of clarity following notations are used: the existing E/E architecture consists of several ECUs (ECU1, ECU2, etc.) connected via multiple communication buses (denoted Bus1, Bus2, etc.) and one or multiple gateways. The new communication that has to be integrated into the existing network is assigned to a distributed function denoted F. F consists of multiple Software Components (SW-Cs) which are mapped on

one or multiple ECUs. Each SW-C has its own communication interfaces through which it sends or receives information, i.e. communication signals packet in PDUs/frames.

A systematic approach for this use case is depicted in figure 6.3. This use case typically consists of the following steps:

1. The Network Data Engineer maps the new communication to the existing PDUs/frames according to the timing parameters defined in the specification of F and the sender-receiver relation between the SW-Cs of F.
2. Depending on the links between the SW-Cs of F it might be necessary to additionally route PDUs/frames between different buses within the network. This happens when the SW-Cs of F are mapped to ECUs that are connected to different buses, e.g. on ECU1 on Bus1 and on ECU2 on Bus2, see also Task "Create Implementation".
3. The Timing Engineer carries out the following analysis steps:
   (a) Analysis 1: The bus load analysis describes the average use of the bus bandwidth. It therefore has to consider the additional traffic generated by the new communication. The bus load analysis must be applied to each bus affected by the new communication and requires the data size and the average timing of the PDUs/frames. The output of the analysis is the timing property GENERIC PROPERTY Load obtained with the timing method GENERIC METHOD Determine Load or specific for CAN buses the timing property SPECIFIC PROPERTY Load (CAN) obtained with the timing method SPECIFIC METHOD Determine Load (CAN). The bus load property is used to initially approve the traffic on each communication bus. The present value of the timing property load obtained for every single bus is compared to the maximum acceptable load on that bus. For typical requirements for the bus load see section 6.3.3. If the bus load exceeds, the communication is not schedulable.
   (b) Analysis 2: In order to validate the network after integrating the new communication, latency requirements have to be also verified on each bus for all PDUs/frames of the legacy and of the new traffic. The latency analysis applies timing methods to compute the timing properties of the PDUs/frames under the resource sharing protocol. The results of the analysis are timing properties such as:
      - response times (including the blocking times due to arbitration) of the PDUs/frames GENERIC PROPERTY Latency obtained with the timing method GENERIC METHOD Determine Latency or specific for CAN buses the timing property SPECIFIC PROPERTY Response Time (CAN) obtained with the timing method SPECIFIC METHOD Determine Response Time (CAN) or
      - the jitter of the PDUs/frames.

The values of the timing properties are compared to the defined requirements and to the previous values of the timing properties. For typical requirements of the PDU/frame response times see section 6.3.3.

(c) Analysis 3: In case that the PDUs/frames associated to the new communication are routed by one or more gateways, the routing times are relevant for the end-to-end timing. The timing method SPECIFIC METHOD Determine Response Time (Routing) applied to the routed PDUs/frames provides the timing properties SPECIFIC PROPERTY Response Time (Routing) due to routing engine. The routing response time is strongly connected to the memory resource requirements for buffering, which need to be considered, but are outside the scope of this document. The values obtained for these properties are compared to the defined requirements. For typical requirements of the routing times see section 6.3.3.

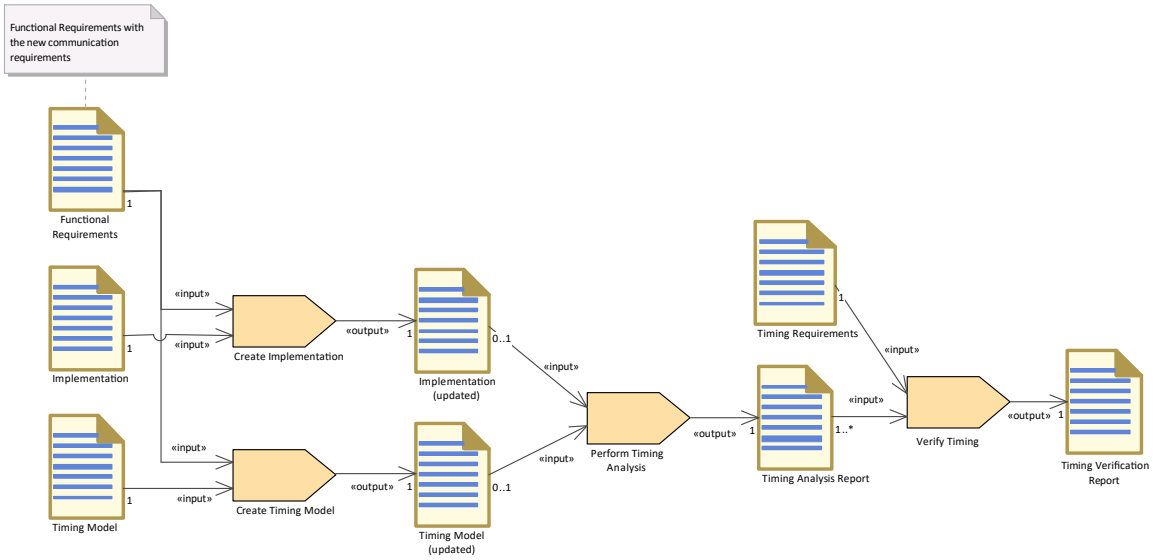

Figure 6.3: SPEM process model for ECU use case "Integration of new communication"

6.3.2 Alternative Scenario

At step #1 of the main scenario, if the new communication exceeds the size of the unused space in the existing PDUs/frames, new PDUs/frames are defined according to the timing parameters of the signals. The impact of the new traffic on the existing communication must be minimized. The methodology continues with Step 2 in the Main Scenario.

6.3.3 Performance/Timing Requirements

The maximum load on each bus shall not exceed a certain bound.

For each frame/PDU, the worst-case response time shall not exceed a certain bound, for example given by the timing requirements. Typically, the cycle time of the frame is used as a bound on the worst-case response time. Otherwise there is a risk for data loss.

Routing times in gateways have to be short. Typically, for each frame/PDU the routing time shall only contribute a minor part to the overall delay. The concrete value depends on the specific functional requirements.

6.4 NW use case "Design and configuration of a new network"

Goal In Context:	Design and feasible integration of a (domain specific) network into existing automotive platform architecture. Possible variants: • New design of the (on-board network) (total automotive network) • Replacement of an existing partial network by a new partial network under use of unaltered legacy ECUs (beside the network connectors). This network is connected to the residual on-board network by a gateway.
Brief Description:	Regarding an existing E/E automotive architecture consisting of several ECUs connected via several legacy networks, it is required to design and to integrate a new designed network (e.g. our active steering example, compare see figure 1.2). The new designed network shall be connected to the residual on-board network via a gateway (for instance to the body or the infotainment domain). Therefore the intra-communication within the new network and the inter-communication between different networks have to be considered. Further, this new network shall be stable extensible in a-priori predictable way, i.e. it shall be possible to analyze the network with respect to all present and future timing requirements. The new network implements communication protocols (e.g. CAN, LIN, Flexray, etc.) and possesses sufficient bandwidth to cover all communication requirements. The communication on the network is specified by a communication matrix containing the PDUs/frames/packages with their protocol specific parameters and the communication behavior (timing parameters).
Scope:	System
Frequency:	Rarely

Precondition:	For the new communication following properties are defined: • The size of the communication signals (SW-C Template / GenericStructureTemplate). • The transmitter and receiver nodes / system mapping • The PDU/frame/package timing/triggering • Required bandwidth • The residual on-board network including gateways and communication matrix Additionally, a set of timing requirements is defined for the communication on the network: • Maximum load on each network • Maximum latency (e.g. response times, routing times) for each PDU/frame/package Furthermore, a specification of the communication paradigm for the existing network controllers is defined, e.g. the CAN controller sends PDUs/frames with different identifiers via a queue (priority ordered or FIFO), while different instances of the same PDU/frame are sending via a register (always send the newest PDU instance). It is assumed that the current (residual) on-board network configuration satisfies the timing requirements.
Success End Condition:	The communication on the new (partial) network was completely defined and the timing requirements of the on-board network are satisfied.
Failed End Condition:	The new communication cannot be defined without violating at least one timing requirement of the on-board network.
Actor(s):	Timing Engineer, Network Data Engineer, E/E Architect

Table 6.3: Characteristic Information of NW UC "Design and configuration of a new network"

6.4.1 Main Scenario

A systematic approach for this use case is depicted in figure 6.4. This use case typically consists of the following steps:

1. The E/E Architect chooses an appropriate network technology to fulfil the communication requirements of the new functions. The consequences for the residual system have to be considered because many ECUs should not be altered if possible.
2. The E/E Architect defines and/or designs the connection point(s) to the residual on-board network (via transparent gateways).
3. The E/E Architect connects the ECUs to the new network and partitions the functions onto these ECUs.
4. The Network Data Engineer collects the data size and the timing requirement for the communication according Task "Collect Timing Requirements".

5. The Network Data Engineer maps the new traffic according to the timing information and the transmitter/receiver relation.
6. Depending on the sender/receiver relation it might be necessary to additionally route PDUs/frame on several networks and gateways.
7. The Timing Engineer creates an analyzable timing model using Task "Create Timing Model".
8. Different types of model-based analysis Task "Perform Model-Based Timing Analysis" shall be carried out:
   (a) Analysis 1: Load analysis determines the average and the maximum use of the network bandwidth and the input buffers of the ECUs. The load analysis must consider the total traffic on the new partial network and on the legacy on-board network as well. Thus, the result of a load analysis, which applies a timing method GENERIC METHOD Determine Load, is the timing property load GENERIC PROPERTY Load. Specifically for CAN buses, the timing method SPECIFIC METHOD Determine Load (CAN) provides as a result the timing property SPECIFIC PROPERTY Load (CAN). The timing property load is used to initially approve the chosen function mapping and architecture and if the new infrastructure is sufficient to cover the communication requirements in general. The present value of the timing property load for every single network is compared to the maximum acceptable load for this network.
   (b) Analysis 2: A detailed latency analysis of all PDUs/frames/packages and every communication relations on the networks is necessary. A timing method such as GENERIC METHOD Determine Latency yields timing properties such as GENERIC PROPERTY Latency and GENERIC PROPERTY Response Time or, specifically for CAN buses the timing property SPECIFIC PROPERTY Response Time (CAN) obtained with the timing method SPECIFIC METHOD Determine Response Time (CAN). Other timing properties such as the jitter or the blocking time are of interest and require corresponding timing methods. Every communication relation has to fulfil its corresponding latency requirement.
   (c) Analysis 3: For network traffic that is exchanged with other networks, the response time requirements for the complete communication event chain have to be considered. For all routers/gateways connected to the network the routing response times may be effected by the routing requirements for the new network. The timing method SPECIFIC METHOD Determine Response Time (Routing) can be use to obtain the timing properties SPECIFIC PROPERTY Response Time (Routing) to verify the network design.
9. Optimization of the design of the new network subject to the requirement to reduce resource needs, to increase system stability and robustness and to allow easily future extensions. A detailed description of the optimization process can be found in NW use case "Optimizing the communication timings".

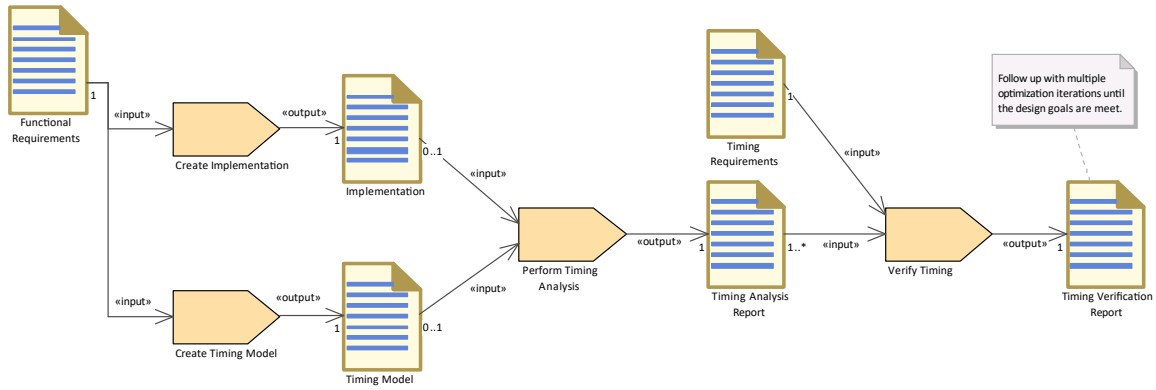

Figure 6.4: SPEM process model for ECU use case “Design and configuration of a new network”

6.4.2 Performance/Timing Requirements

The maximum load on each bus shall not exceed a certain bound.

For each frame/PDU, the worst-case response time shall not exceed a certain bound, for example given by the timing requirements. Typically, the cycle time of the frame is used as a bound on the worst-case response time.

Routing times in gateways have to be short. Typically, for each frame/PDU the routing time shall not exceed 10% of the cycle time of the frame.

6.5 NW use case “Remapping of an existing communication link”

This use case focuses on remapping an existing communication link within an existing network.

Goal In Context:	Validate the communication on the network after reconsidering the mapping of an existing communication link.
Brief Description:	Assuming an E/E automotive architecture that contains ECUs connected via one or more buses, it is required to remap an existing communication link to a new resource within the network (e.g.mapping the motor control signal from CAN to FlexRay assuming that the electric motor is directly connected to FlexRay, see figure 1.2). The buses within the network may implement different communication protocols (e.g. CAN, LIN, Flexray). The communication on each bus is specified by a communication matrix containing the PDUs/frames with their protocol specific parameters and the communication behavior (timing parameters, e.g. 10ms maximum latency for the motor control signal).
Scope:	System
Frequency:	Regular

Precondition:	The signals describing the communication link is known and included in the communication matrix. Additionally, for the communication on the network is defined a set of timing requirements: • Maximum bus load on each bus • Maximum latency (e.g. response times, routing times) for each communication frame. Furthermore, the specification of the communication paradigm for the existing bus controllers is available. For example, the CAN controller sends CAN message frames with different identifiers via a queue (priority ordered or FIFO), while different instances of the same frame are sent via a register (always send the newest instance of the frame). It is assumed that the current network configuration satisfies the timing requirements.
Success End Condition:	The communication on the network after function remapping fulfils the timing requirements. The communication matrix needs to be updated.
Failed End Condition:	The communication on the network after function remapping cannot be defined without violating at least one timing requirement.
Actor(s):	Timing Engineer, Network Data Engineer

Table 6.4: Characteristic Information of NW UC "Remapping of an existing communication link"

6.5.1 Main Scenario

For the sake of clarity following notations are used: the communication link to be remapped is currently assigned to Bus1. The resource that will host the communication link after remapping is denoted Bus2.

A systematic approach for this use case is depicted in figure 6.5. This use case typically consists of the following steps:

1. The Network Data Engineer identifies the PDUs/frames on Bus1 assigned to the communication link. These must be transmitted on Bus2 after remapping the communication link.
2. The PDUs/frames assigned to the communication link and additionally required by other links on Bus1 must be routed on Bus2 after remapping the communication link to Bus2. Otherwise, in case that these PDU/frame are not required by other nodes at Bus1, one may decide to remove them from Bus1.
3. The PDU/frames moved or copied to Bus2 should preserve the parameters of the communication protocol defined for Bus1, in order to ensure the function compatibility with the different architecture variants.
4. PDUs/frames required by the communication link at Bus2 and that are not originally sent by another sender on the Bus2 need to be routed/transmitted to Bus2.
5. The Timing Engineer carries out the following analysis steps:

(a) Analysis 1: The bus load analysis describes the average use of the bus bandwidth. The analysis has to consider the additional traffic on Bus2 after remapping the communication link to Bus2. The analysis requires the data size and the average timing of the PDUs. The output of the analysis which applies a method GENERIC METHOD Determine Load or, SPECIFIC METHOD Determine Load (CAN) for CAN buses, is the static bus load captured by a timing property GENERIC PROPERTY Load or, SPECIFIC PROPERTY Load (CAN). The bus load property is used to initially approve the traffic on Bus2. Optionally, one can carry out bus load analysis on Bus1 to determine the freed performance slack after remapping the communication link to Bus2. The value of the timing property load obtained for every single bus is compared to the maximum acceptable load on that bus. For typical requirements for the bus load see section 6.5.2. If the bus load exceeds the communication is not schedulable.

(b) Analysis 2: In order to validate the communication on the network after remapping the communication link to Bus2, the latency requirements of the PDUs/frames on Bus2 must be verified. The latency analysis of the PDUs/frames computes the timing properties of the PDUs/frames under the resource sharing protocol. The results of the analysis, which applies a timing method GENERIC METHOD Determine Latency or SPECIFIC METHOD Determine Response Time (CAN) for CAN buses, are timing properties response times of the PDUs/frames such as GENERIC PROPERTY Latency / GENERIC PROPERTY Response Time or SPECIFIC PROPERTY Response Time (CAN) in case of CAN buses. Other timing properties such as the jitter of the PDUs/frames or the blocking times due to arbitration are of interest and require corresponding timing methods. The values of the timing properties are compared to the specified requirements. For typical requirements of the PDU/frame response times see section 6.5.2.

(c) Analysis 3: In case that the PDUs/frames required at Bus2 are routed by one or more gateways, the routing times are relevant for the end-to-end timing. The timing method SPECIFIC METHOD Determine Response Time (Routing) applied to the routed PDUs/frames provides the timing properties SPECIFIC PROPERTY Response Time (Routing) due to routing engine. The routing response time is strongly connected to the memory resource requirements for buffering, which need to be considered, but are outside the scope of this document. The values obtained for these properties are compared to the specified requirements. For typical requirements of routing times see section 6.5.2.

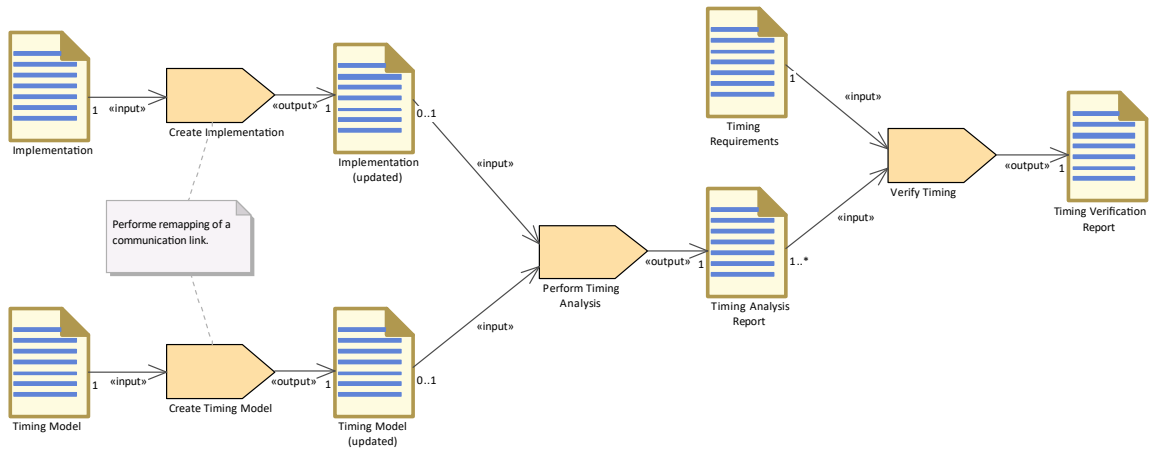

Figure 6.5: SPEM process model for ECU use case "Remapping of an existing communication link"

6.5.2 Performance/Timing Requirements

The maximum load on each bus shall not exceed a certain bound.

For each frame/PDU, the worst-case response time shall not exceed a certain bound, for example given by the timing requirements. Typically, the cycle time of the frame is used as a bound on the worst-case response time.

Routing times in gateways have to be short. Typically, for each frame/PDU the routing time shall not exceed for example 10% of the cycle time of the frame.

6.6 NW use case "Changes of the E/E-Topology"

This use case becomes applicable, if it becomes necessary to change the E/E-Topology or different options for E/E-Topologies need to be evaluated. The E/E-Topology impacts the timing behavior and it has to be verified with a timing analysis of the E/E-Topology, that all communication links affected by the topology change still fulfill their timing requirements.

Since it can be difficult and costly to change the E/E-Topology at a late stage during development, especially if already produced hardware has to be replaced, it is good practice to use model-based timing analysis for the evaluation of E/E-Topology design options. Later on for the verification of small optimizations of an E/E-Topology, the timing analysis may be directly performed on the implementation.

For defining the scope of the timing analysis and setting focal points for the timing model, it is important to understand how different topology changes impact the timing behavior:

6.6.1 Moving a single ECU

The simplest modification to the E/E-Topology is moving an ECU to another network. Motivation for this could be to move the ECU closer to a time critical communication partner or to reduce the routing requirements by placing the ECU on the same networks as most of its communication heavy communication partners. By move the ECU some communication path become shorter, but other will become longer. So it is important that all messages exchanged with this ECU still fulfill the latency timing requirements. On the other network the load can be increased by additional messages from the ECU routed through these networks, this can negatively impact the response time of already existing messages on these networks. The changed location can also impact routing paths, resulting in the traffic being routed through a previously unrelated network affecting it load and timings. The new routing paths can also affect the routing performance of gateways. It may increase the blocking time of message routed through the same ports and worst case can impact the routing performance of the whole gateway, affecting all messages routed by the gateway.

6.6.2 Moving multiple ECUs

Moving multiple ECUs to another network. The goal of this change can be to rebalanced the load of two networks. If there is a group of ECUs with heavy communication and on another network the load is still low. It can be on option to move the ECUs to this network. It has to be taken into account that the load will not only be increased by the communication between these ECUs but also the communication to other ECUs that now has to get routed through this network. The additional messages will impact the latency timing, jitter and blocking time for existing messages. And as described for the single ECU impact the timing from the new routing requirements.

6.6.3 Adding a network

Adding a new network to the E/E-Topology. If the network load of one or more networks exceeds the maximum acceptable load or the load negatively impacts the response times on these networks, it may be required add an additional network to the E/E-Topology and move suitable ECU from these networks to the new network. The suitability of ECUs is determined by the communication requirements. The goal should be to place ECUs which exchange time critical or a lot of data on the same network. Assuming the new network is connected to the same gateway and the gateway is able to handle the additional routing effort easily, the overall timing performance will increase for the previously overloaded networks. This is the case, if the additional routing time at the gateways is smaller, than the blocking time was on the previously overloaded network. If the routing performance of the gateway is heavily affected by the change or the new network is located further away, requiring routing through other networks, the timing analysis will be more complex. It will need to be performed for all parts affected by the new routing requirements.

6.6.4 Removing a network

Removing a network from the E/E-Topology. If the currently available network resources are heavily under utilized, it can be an option to reduce the complexity and costs of the E/E-Topology by removing a network. The ECUs connected to this network need than to be connected to other networks. For ECUs with multiple network connections it may also be an option to re-route it traffic through the another network connection. If the removal of the network is done by merging one network into the other, the only impact on the routing will be the blocking time at the gateway port connecting the combined network. It can even decrease, if a lot of the traffic was between the merged networks. A part from this, only the combined network needs to be analyzed, to verify that all timing requirements are still met. If the ECUs from the removed are distributed to other networks or the traffic is routed through other connections, the new routing requirements can have a wide ranging impacts on the timing behavior as described above.

Goal In Context:	Verify the feasibility of changes of the E/E-Topology.
Brief Description:	Changing the E/E-Topology can have a wide influence on the timing behavior of the communication. It has to be confirm through timing analysis, that the changed E/E-Topology still fulfills all timing requirements.
Scope:	System
Frequency:	Rarely
Precondition:	Description of the changed E/E-Topology with all ECUs and the networks connecting the ECUs. A specification of the routing strategies for the gateways connecting the networks is available. The communication paradigms for the used network controllers are specified, e.g. the CAN controller sends PDUs/frames with different identifiers via a queue (priority ordered or FIFO), while different instances of the same PDU/frame are sending via a register (always send the newest PDU instance). Additionally, for the communication on the network a set of timing requirements is defined: • Maximum load all relevant networks • Maximum latency (e.g. response times, routing times) for all relevant messages.
Success End Condition:	The changed E/E-Topology fulfills all timing requirements.
Failed End Condition:	At least one timing requirement is violated by the changed E/E-Topology.
Actor(s):	Timing Engineer, Network Data Engineer, E/E Architect

Table 6.5: Characteristic Information of NW UC “Changes of the E/E-Topology”

6.6.5 Main Scenario

A systematic approach for this use case is depicted in figure 6.6. This use case typically consists of the following steps:

1. The E/E Architect provides a design for the changed E/E-Topology.

2. The Network Data Engineer maps the traffic according to the timing information and the sender/receiver relation.
3. Depending on the sender/receiver relation it might be necessary to additionally route messages on several networks and gateways.
4. The Network Data Engineer updates the routing tables for the changed E/E-Topology.
5. The Timing Engineer updates timing model to match the changed E/E-Topology using Task “Create Timing Model”.
6. The Timing Engineer carries out the following analysis steps:
   (a) Analysis 1: The network load analysis describes the average use of the network bandwidth. The analysis has to consider the traffic on the networks affected by the changes to the E/E-Topology. The analysis requires the data size and the average timing of the messages. The output of the analysis which applies a method GENERIC METHOD Determine Load, is the static network load captured by a timing property GENERIC PROPERTY Load. The network load property is used to initially approve the traffic on the affected networks. It allows to verify that the maximum acceptable load is not exceeded on the changed networks or can show freed up resources on the networks. For typical requirements for the bus load see section 6.6.6. If the network load exceeds the constrains the changes to the E/E-Topology are not viable.
   (b) Analysis 2: In order to validate the communication on the network after changes of the E/E-Topology, the latency requirements of all affected messages must be verified. The latency analysis of the messages computes the timing properties of the messages under the resource sharing protocol. The results of the analysis, which applies a timing method GENERIC METHOD Determine Latency, are timing properties of the messages such as GENERIC PROPERTY Latency or GENERIC PROPERTY Response Time. Other timing properties such as the jitter of the messages or the blocking times due to arbitration are of interest and require corresponding timing methods. The values of the timing properties are compared to the specified requirements. For typical requirements of the messages response times see section 6.6.6.
   (c) Analysis 3: Changes of the E/E-Topology impact the routing. The routing time analysis of the routed messages provides the delay values due to routing engines. These usually consist of buffering delay and arbitration delay. The results of the routing time analysis are the routing response times, the blocking times due to buffering and arbitration, or the memory requirements for buffering. The values obtained for these properties are compared to the specified requirements. For typical requirements of routing times see section 6.6.6.

7. The E/E Architect decides, if further optimizations of the design of the E/E-Topology are required. The optimization should be subject to the requirement to reduce resource needs, to increase system stability and robustness and to allow easily future extensions. A detailed description of the optimization process can be found in NW use case "Optimizing the communication timings".

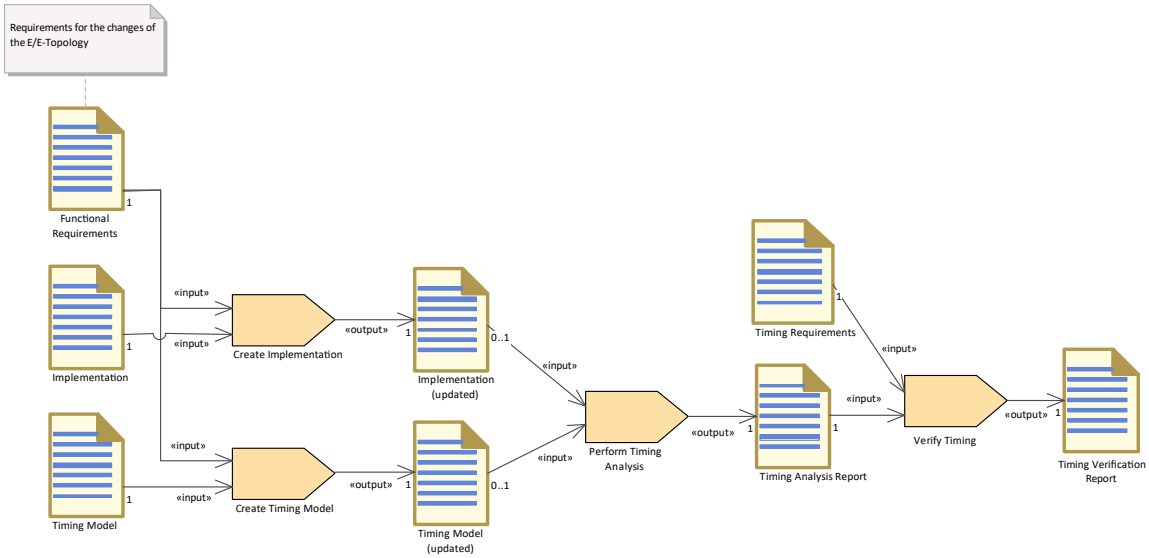

Figure 6.6: SPEM process model for NW use case "Changes of the E/E-Topology"

6.6.6 Performance/Timing Requirements

The maximum load on each bus shall not exceed a certain bound.

For each frame/PDU, the worst-case response time shall not exceed a certain bound, for example given by the timing requirements. Typically, the cycle time of the frame is used as a bound on the worst-case response time.

Routing times in gateways have to be short. Typically, for each frame/PDU the routing time shall not exceed for example 10% of the cycle time of the frame.

Changing the E/E topology has a direct impact on the end-to-end timings. For more details on the verification of end-to-end timing requirements refer to the appropriate use cases in chapter End-to-End Timing for Distributed Functions.

6.7 NW use case "Optimizing the communication timings"

The optimization of the communication timing properties is a generic use case, that can come up frequently during the design or maintenance of a communication network. It may be required within the context of other use cases (e.g. NW use case "Remapping of an existing communication link"), if a new or updated design does not meet the timing

requirements, after identifying a timing violation or as a prerequisite to free up design space for a modification.

Based on the motivation to perform the optimization quantifiable goals must be defined. These optimization goals allow to measure the success of the changes performed for optimization. E.g. if the responsiveness of the system shall be increased, a lower response time for a certain message can be the optimization goal.

Goal In Context:	Remove timing violations or fulfilling timing optimization goals
Brief Description:	Based on timing requirements, while taking all timing constraints into account the overall timing properties for a communication network is optimized.
Scope:	System
Frequency:	Regular
Precondition:	A set of timing requirements is defined for the communication on the network: • Maximum load on each network • Maximum latency (e.g. response times, routing times) for each PDU/frame/package Furthermore, a specification of the communication paradigm for the existing network controllers is defined, e.g. the CAN controller sends PDUs/frames with different identifiers via a queue (priority ordered or FIFO), while different instances of the same PDU/frame are sending via a register (always send the newest PDU instance). Optionally: Additional timing optimization goals are defined.
Success End Condition:	The communication fulfils the timing requirements and optimization goals.
Failed End Condition:	The communication on the network is violating at least one timing requirement or any optimization goal is not fulfilled.
Actor(s):	Network Data Engineer, Timing Engineer

Table 6.6: Characteristic Information of NW UC "Optimizing the communication timings"

6.7.1 Main Scenario

A systematic approach for this use case is depicted in figure 6.7. This use case typically consists of the following steps:

1. The use case begins when the Network Data Engineer becomes aware of timing violations or the need to add more functionality into an already heavily loaded system.
2. The Timing Engineer analyzes the current system under the condition that lead to the timing violation or analyzes the current system and to find hot-spots. These are situations in the schedule, where either timing requirements (e.g. worst case response time or jitter) or resource consumption constraints (e.g. bus load limit) are violated already or would be if more load was added.
3. Exploration of available options in order to relax the hot-spots. Often there are multiple option available to resolve an issue. E.g. if for a frame the worst-case response time is exceeded, a possible solution could be to increase the priority

of the frame. Another option may be to improve the response time by reducing the bus load, if communication can be remapped to another communication link.

4. The Network Data Engineer performs a trade-off analysis, to weight the different possibilities for the optimization of the timing and its impact on the system. Usually a modification comes with a cost attached. E.g. increasing the priority of a frame, results in decreased priority of other frames, which can causes an increase in their response time.
5. The Network Data Engineer decides for a modification and changes the timing-model/the network configuration. The analysis may reveal timing issues due to the performance of the router/gateway, which may require the replacement with a more potent hardware or an optimization of the routing configuration (see: Scheduling and Sporadic Events)
6. The Timing Engineer verifies the timing of the communication network by performing Task "Verify Timing"
7. The verification may reveals that the timing violations were not resolved, or that new timing violations were introduced. In this case, a new optimization iteration can be started from step 3 using the Timing Analysis Report from step 6.

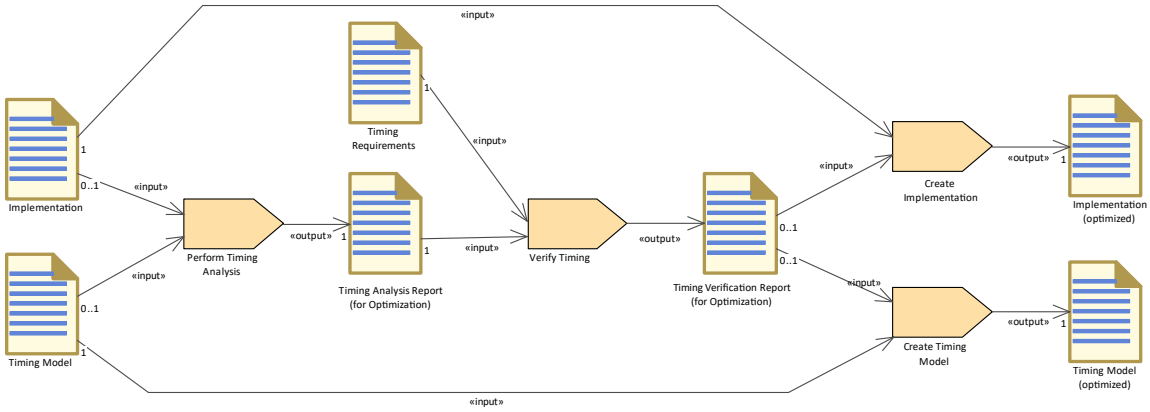

Figure 6.7: SPEM process model for ECU use case "Optimizing the communication timings"

6.7.2 Performance/Timing Requirements

The maximum load on each bus shall not exceed a certain bound.

For each frame/PDU, the worst-case response time shall not exceed a certain bound, for example given by the timing requirements. Typically, the cycle time of the frame is used as a bound on the worst-case response time.

Routing times in gateways have to be short. Typically, for each frame/PDU the routing time shall not exceed for example 10% of the cycle time of the frame.

6.8 NW use case "Derive timing properties of a message on a network segment"

Timing properties of a message are of interest when creating a timing model or for verification of timing requirements for early design decisions.

The timing properties can be acquired through timing analysis of an implementation of the network for which the timing model shall be created or by analyzing the timing of an implementation that is very similar. When acquiring timing properties from a reference implementation, it is important to note that the timing properties are usually influenced by other messages send on the same network segment. So not only the reference message of the analyzed implementation need to be similar, but also the other messages on the network segment. For messages with time deterministic behavior, for example frames from the static segment from the Flexray protocol or unconditional frames from the LIN protocol, the timing properties can be derived directly from the Communication Matrix (see table 10.12).

Goal In Context:	Derive timing properties of a message on a network segment.
Brief Description:	Derive timing properties of a message from a timing analysis of a reference implementation or Communication Matrix (see table 10.12).
Scope:	Network Segment
Frequency:	On request
Precondition:	A Communication Matrix (see table 10.12) with all relevant information of the message is available. A Timing Analysis Report (see table 10.12) over all relevant scenarios for the message of interest.
Success End Condition:	All required timing properties of the message are known.
Failed End Condition:	At least one required timing property of the message is unknown.
Actor(s):	Network Data Engineer, Timing Engineer, E/E Architect

Table 6.7: Characteristic Information of NW UC "Derive timing properties of a message on a network segment"

6.8.1 Main Scenario

A systematic approach for this use case is depicted in figure 6.8. This use case typically consists of the following steps:

1. The E/E Architect or Timing Engineer requires information on the timing properties of a message on a network segment.
2. Depending on wether the message is time deterministic or not:
   (a) The Network Data Engineer derives the timing properties from the Communication Matrix (see table 10.12).
   (b) The Timing Engineer derives the timing properties from the Timing Analysis Report (see table 10.12).

3. Optionally: The Network Data Engineer adjusts the timing properties (e.g. add safety margins) to compensate for difference between the reference implementation and the target system.

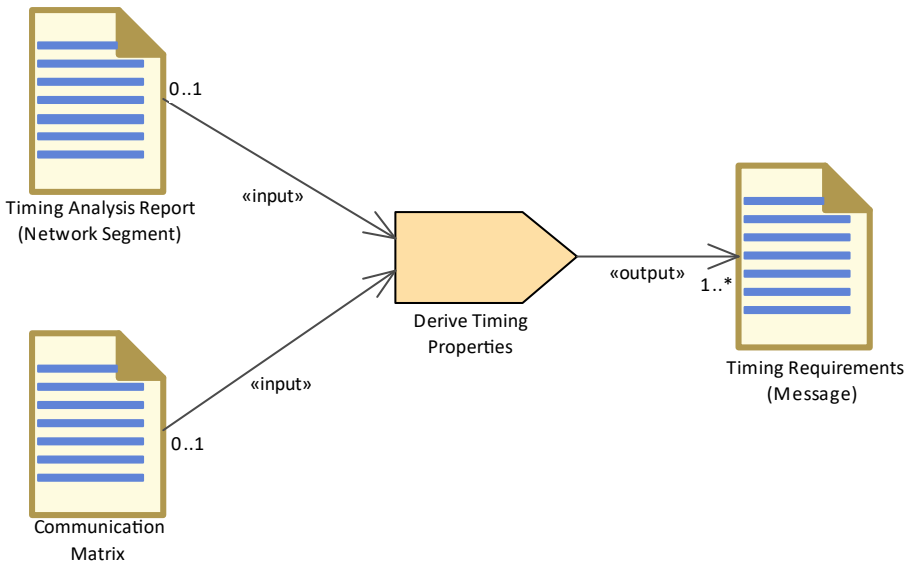

Figure 6.8: SPEM process model for NW use case "Derive timing properties of a message on a network segment"

7 Timing for SW-Integration on ECU Level

This chapter outlines use cases relevant for software integration into a single ECU with respect to timing issues. Network related aspects are covered by chapter 6 and have only an indirect impact on the timing on the ECU level. On the ECU level, the scheduling of tasks and interrupts together with the execution times of the various code fragments define the timing behavior of the overall software for this specific ECU. Depending on the scheduling and the execution times, given deadlines are met or missed. The use cases in this chapter help to solve problems or tasks which are related to scheduling and/or execution times.

Although speaking of “ECU-level”, it is important to bear in mind a single ECU can come with multiple processors each of which comes with its own scheduling. Even multiple cores on one processor are seen more and more often [16]. However, the principles in this chapter still remain valid and can be reflected on each “scheduling entity” (=core).

Typical terms used in this chapter are:

- Execution Time (e.g.: CET, BCET, WCET..), see section 2.2 and 9.4.
- CPU-Load , see section 9.4.
- Interrupt Load, see section 9.4.
- Response Time, see section 9.4.
- Latency, see section 9.4.

7.1 Platform Specific Terminology

Pending on the AUTOSAR platform used on the ECU the terminology to describe the software is different. To avoid doubled description for the Classic- and Adaptive Platform, we use generic terms as much as possible to describe the different use-cases. Table 7.1 provides a mapping from generic term to platform specific term to help users of the different platforms understand the meaning behind the generic terms used in this chapter.

Generic	Adaptive	Classic
ECU	Machine	ECU
Software Entity (SWE)	Adaptive Application, Functional Cluster	BSW Module, SWC, CDD
Schedulable Entity	Thread	Task, Interrupt
Exclusive Area	n.a. (not specified for AP, but the implementation will also contain code sections that are protected against parallel execution)	Exclusive Area

Table 7.1: Mapping of generic terms to platform specific terms

7.2 Example

In the introduction a top-level-example for timing is given in figure 1.2 on page 16. The ECU-level deals with a fragment of the top-level-example, namely the scheduling aspects and code execution aspects of the ECUs involved, see figure 7.1. The use cases of this chapter will refer to this example.

The example shows the scheduling and code execution of a CP ECU, but does not limit the generality of tracing the usage of any compute resource on any platform.

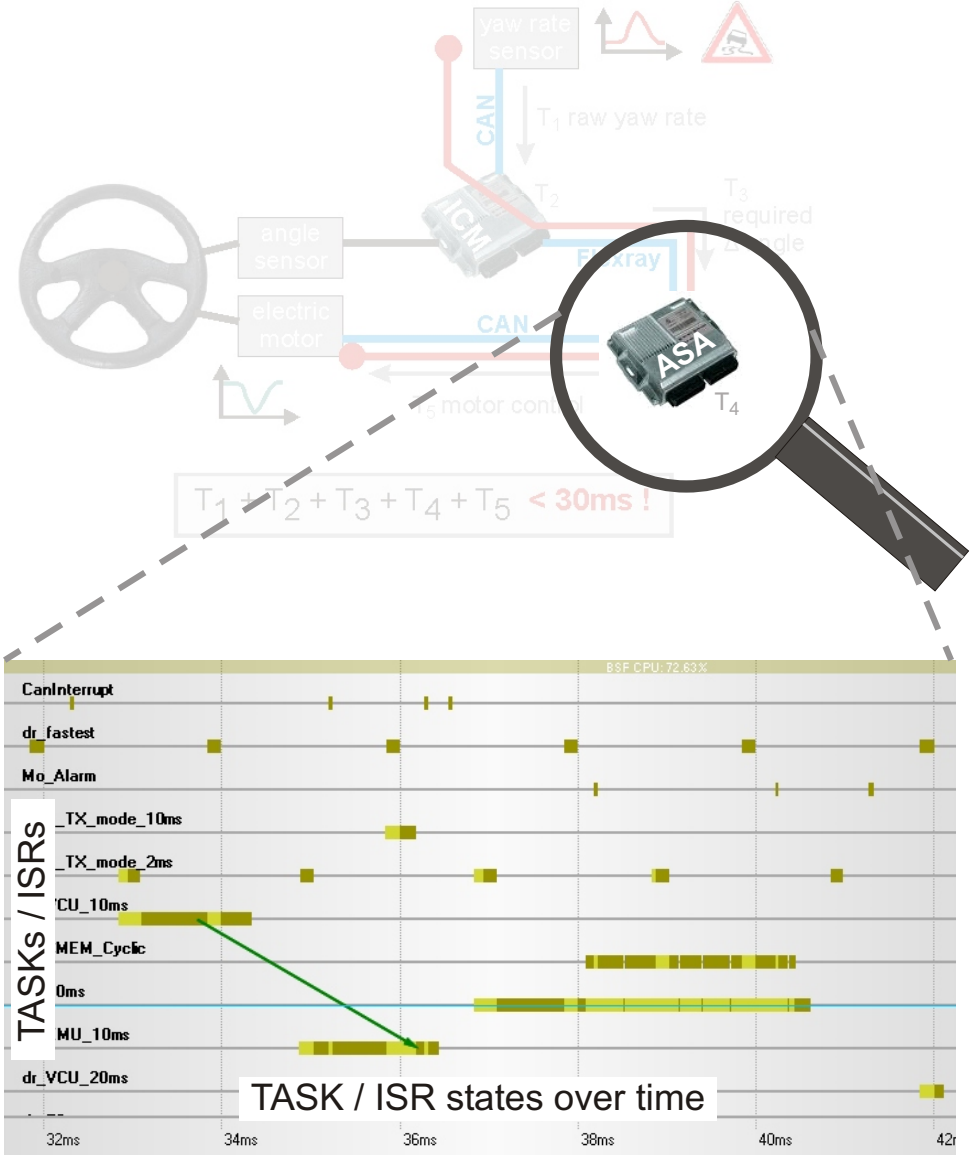

Figure 7.1: Focus of this chapter: scheduling and code execution time inside ECUs

7.3 Overview of ECU Use Cases

Different phases/use cases in the development of a vehicle system shall be considered, which are listed in Table 7.2. Figure 7.2 gives an overview.

Section	Use Case	Page
7.4	ECU use case “Create Timing Model of the entire ECU”	106
7.5	ECU use case “Collect Timing Information of a SWE”	109
7.6	ECU use case “Derive timing properties of an executable entity”	111
7.7	ECU use case “Verification of Timing”	112
7.8	ECU use case “Debug Timing”	114
7.9	ECU use case “Optimize Timing of an ECU”	117
7.10	ECU use case “Optimize Scheduling”	119
7.11	ECU use case “Optimize Code”	122
7.12	ECU use case “Integrate a new function”	123

Table 7.2: List of ECU specific use cases

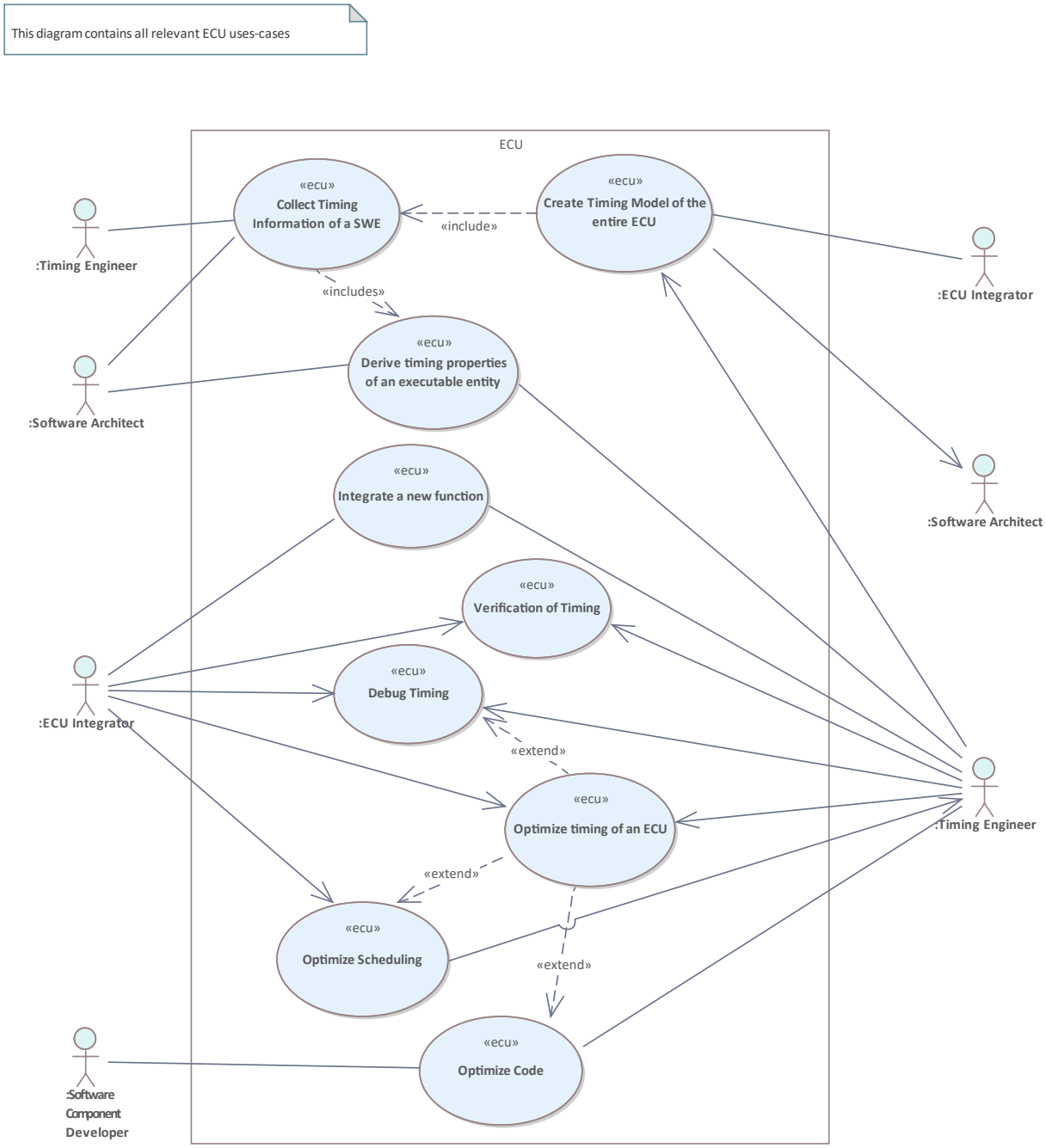

Figure 7.2: Use case diagram: Timing Analysis for ECU

7.3.1 Assumptions

If not otherwise stated the following assumptions hold true for all use cases described in this chapter:

For CP ECUs:

1. The ECU Extract for a specific ECU is available including the ECU Extract content for System Timing.
2. The VFB View (SW-C Template, hierarchy of SW-Cs) of all SW-Cs mapped onto the specific ECU is available.
3. SW-C descriptions are available
4. The interaction takes place between one OEM and one tier1 supplier
5. All SW-Cs including C source code and object files are available.
6. All required BSW modules are available including C source code, object files and ECU configuration.
7. RTE can be generated
8. The contents of this chapter deal solely with the subject matter timing analysis. The assumption made is that any “system” subject to timing analysis is valid from the functional point of view.

For AP ECUs:

1. The Machine Manifest for a specific ECU is available.
2. The Execution Manifest with all execution constrains described for all Adaptive Applications and Functional Clusters mapped onto the specific ECU is available.
3. All Adaptive Executables for the Adaptive Applications included in the ECU are available.
4. All Functional Clusters are fully configured.
5. All Adaptive Applications the required Service Instance Manifests are available.

7.4 ECU use case “Create Timing Model of the entire ECU”

This section describes how to generate a timing model for a complete ECU. The difficulties to describe the use case in a unique manner are justified by the fact that since the OEM and the Tier1 use different abstraction levels and semantics, their views on this use case differ. Especially if they work during different phases in the development process, this effect is reinforced.

Nevertheless, some basic assumptions are valid for all levels of granularity and all development phases.

In the context of the example shown in figure 7.1 on page 103, the creation of timing model means to build up an abstract representation of the timing behavior of the ECU as the system under observation.

As a matter of fact, the creation of a timing model of the entire ECU is one of the important steps to gain a complete system understanding. All other use cases can be

seen as somehow connected use cases, since the existence of a timing model is a precondition in order to execute the steps in other use cases.

A timing model of an ECU collects all timing data such as timing requirements (Task "Collect Timing Requirements"), timing measurements (SPECIFIC PROPERTY Execution Time, GENERIC PROPERTY Latency) and also timing relevant configuration data (such as RTE or BSW configuration) and can be used in other use cases as well. Or in other words: Without the existence of a timing model it is hardly possible to handle the following use cases.

Depending on the development phase, the timing model can be based mainly on assumptions and requirements (requirement timing model) or mainly based on measurements and existing configuration information. Ideally, both views are accessible in one model.

7.4.1 Characteristic Information

Goal In Context:	Collect all relevant timing information for a selected ECU or rather timing model
Brief Description:	Collect all relevant timing information for an ECU and create a timing model of the entire ECU
Scope:	ECU
Frequency:	On request
Precondition:	Knowledge about basic functionality of the ECU and basic understanding about the functional requirements of the ECU and the application domain
Success End Condition:	Timing Model is created and reflects all timing information
Failed End Condition:	E.g. timing information cannot be collected
Actor(s):	ECU Integrator, Software Architect, Timing Engineer

Table 7.3: Characteristic Information of ECU UC "Create Timing Model of the entire ECU"

7.4.2 Main Scenario

A systematic approach for this use case is depicted in figure 7.3 for the classic platform and in figure 7.4 for the adaptive platform. The following steps typically apply:

1. The ECU Integrator and Software Architect provide all available timing data for the specific ECU (Task "Collect Timing Requirements", SPECIFIC PROPERTY Execution Time).
2. The Timing Engineer checks the collected data.
3. The Timing Engineer adds the retrieved timing data to timing model (Task "Create Timing Model").
4. The use case ends with ECU timing model. The timing information will be usable for further work, e.g. Task "Perform Model-Based Timing Analysis".

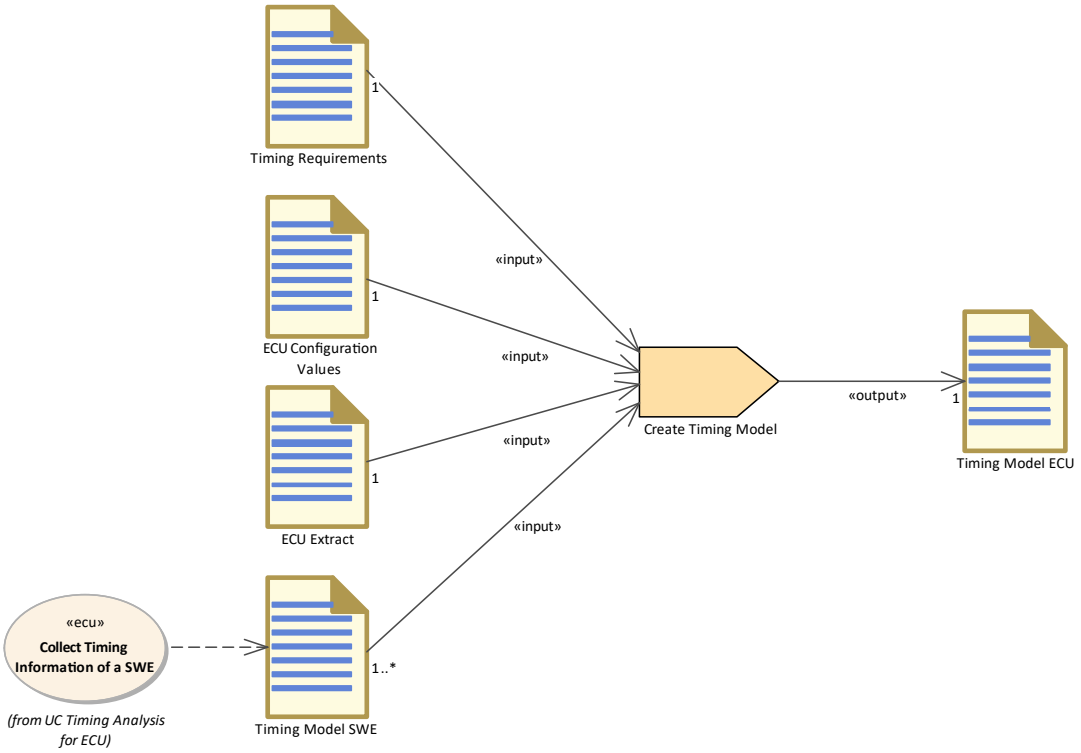

Figure 7.3: SPEM process model for ECU use case "Create Timing Model of the entire ECU CP"

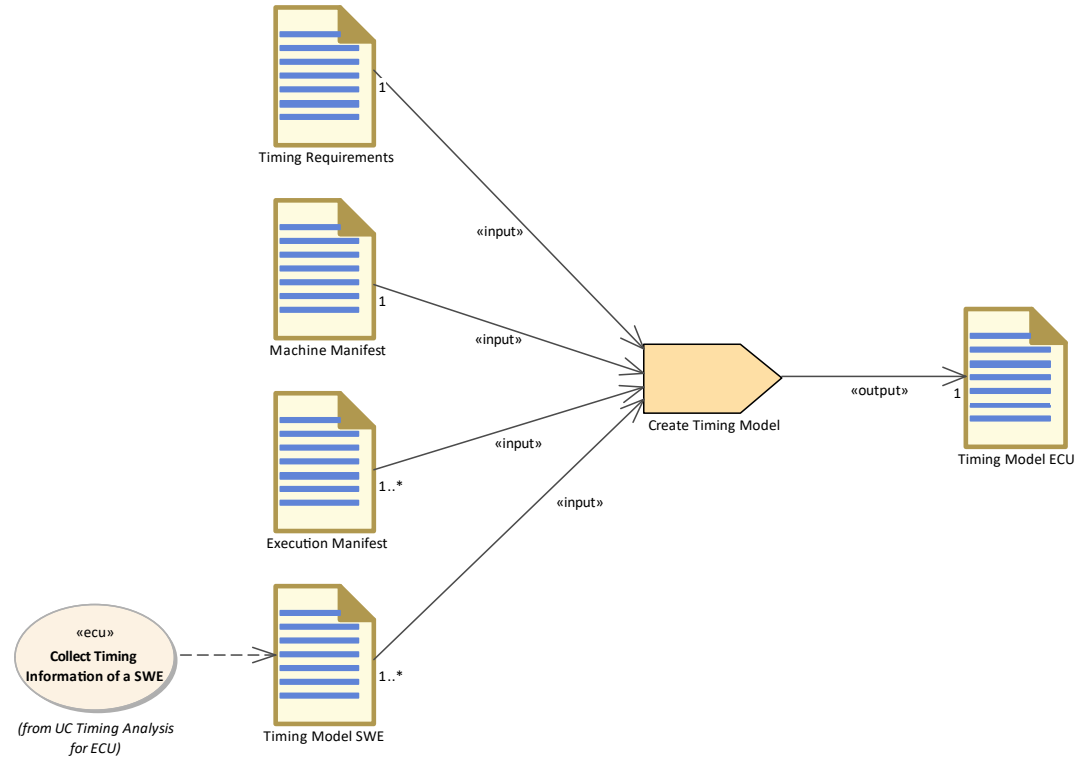

Figure 7.4: SPEM process model for ECU use case "Create Timing Model of the entire ECU AP"

An appropriate tool chain is required. Such a tool chain must be able to import and export the artifacts generated from different tools during the complete development cycle.

7.4.3 Alternative Scenario

Due to the different levels of granularity and different phases different scenario extensions are possible. In concrete cases the Timing Engineer must choose the matching scenario.

7.5 ECU use case "Collect Timing Information of a SWE"

In section 7.4, the creation of a timing model is described. Collecting timing information is required in order to build up a timing model. In this use case, collecting timing information of a specific SWE is described. For the example shown in figure 7.1 on page 103, this could mean getting information about a specific SWE inside the ECU "ASA", e.g. its maximum execution time (SPECIFIC PROPERTY Execution Time).

7.5.1 Characteristic Information

Goal In Context:	Collect all relevant timing information of a selected SWE
Brief Description:	Collect all relevant timing information for a SWE
Scope:	SWE for a specific target
Frequency:	On request
Precondition:	Knowledge about basic functionality of the SWE
Success End Condition:	Timing Model is created and reflects all timing information
Failed End Condition:	E.g. timing information cannot be collected
Actor(s):	Software Architect, Timing Engineer

Table 7.4: Characteristic Information of ECU UC "Collect Timing Information of a SW Entity"

7.5.2 Main Scenario

A systematic approach for this use case is depicted in figure 7.5. The following steps typically apply:

1. The use case begins when the responsible Software Architect begins the collection of timing information which is usually triggered by the request of the ECU Integrator
2. The Software Architect collects all available timing data for the specific SWE.
   - Some estimation about previous and similar project, methods, see section 9.5

- Runtime measurements on runnable level and below, methods, such as Processor-In-The-Loop Simulation (PIL) or Static Worst Case Execution Time Analysis see section 9.5
- Timing requirements for this SWE (Task "Collect Timing Requirements") based on functional requirements, for instance
  - `Trigger` events
  - Latencies
  - Jitters
  - Execution orders
  - Relations to safety-relevant requirements
- The following methods can be used to collect relevant properties (e.g. GENERIC PROPERTY Load, Interrupt Load, SPECIFIC PROPERTY Execution Time, GENERIC METHOD Determine Latency)
  - Tracing
  - Scheduling Analysis
  - Scheduling Simulation

3. The Timing Engineer adds the retrieved timing data to timing model (part of Task "Create Timing Model").
4. The use-case ends with SWE timing information. The timing information will be usable for SWE integration in the overall system.

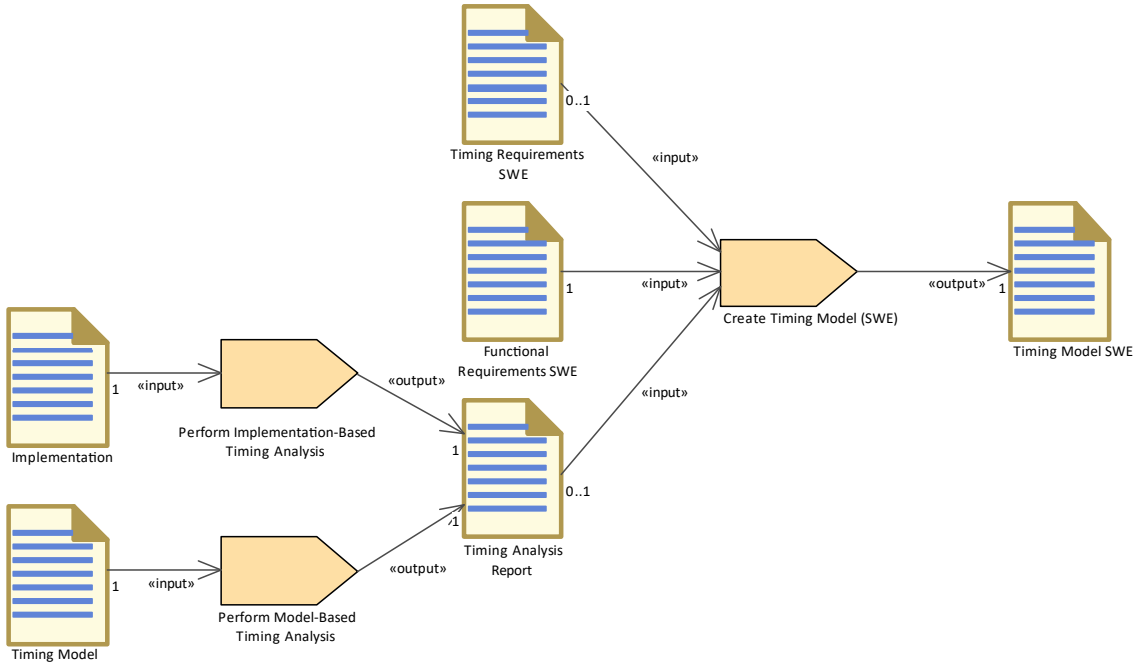

Figure 7.5: SPEM process model for ECU use case "Collect Timing Information of a SWE"

7.5.3 Alternative #1 Scenario

At step #2 of the main scenario the sub-steps can be carried out in arbitrary order or might be skipped. The justification for skipping can be missing information at this specific phase in time.

7.6 ECU use case "Derive timing properties of an executable entity"

In section 7.4, the creation of a timing model is described. Collecting timing information is required in order to build up a timing model. Obtaining timing properties of an executable entity poses many challenges. The timing properties can be strongly influenced by the target platform and the compiler, which can make it difficult to reuse measurements. Additionally increasing complexity in CPU architectures make it difficult to measure or calculate the timing properties. E.g. features like branch prediction cause the same code to have varying runtimes or because of pipelining the instruction order has to be considered when calculating the runtime.

7.6.1 Characteristic Information

Goal In Context:	Derive timing properties of an executable entity
Brief Description:	Derive timing properties of an executable entity from a Timing Analysis Report (see table 10.12).
Scope:	Executable entity on a specific target platform
Frequency:	On request
Precondition:	A working implementation containing the executable entity of interest.
Success End Condition:	Timing properties of the executable entity have been obtained
Failed End Condition:	Timing properties cannot be derived from the Timing Analysis Report (see table 10.12)
Actor(s):	Software Architect, Timing Engineer

Table 7.5: Characteristic Information of ECU UC "Derive timing properties of an executable entity"

7.6.2 Main Scenario

A systematic approach for this use case is depicted in figure 7.6. This use case typically consists of the following steps:

1. The use case begins when the responsible Software Architect begins the collection of timing information. E.g. for the creation of a timing model of the ECU.
2. The Software Architect determines the test conditions under which the required timing properties of the executable entity can be observed. E.g. for acquiring the

WCET of an executable entity, the Software Architect needs to determine the test conditions, which cause the executable entity to execute its critical path.

3. The Timing Engineer uses the test conditions to execute Task "Perform Implementation-Based Timing Analysis".
4. With the Timing Analysis Report (see table 10.12) the Software Architect can perform Task "Derive Timing Properties" to obtain the timing properties of the executable entity.

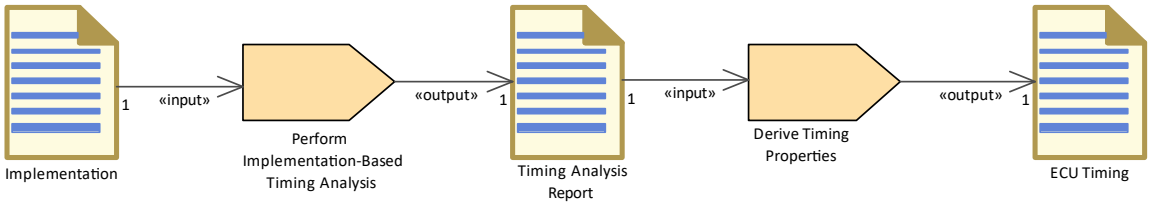

Figure 7.6: SPEM process model for ECU use case "Derive timing properties of an executable entity"

7.6.3 Alternative Scenario 1

It may not be feasible to determine the test conditions under which the required timing properties can be observed. In this case it is an option to use Fuzzing during the timing analysis. This does not guaranty that the correct test conditions are meet and the derived timing properties should only be seen as estimates.

7.6.4 Alternative Scenario 2

In the early stages of development for a new target platform, the hardware or an emulator for the hardware may not be available. In this case an alternative platform can be used for the timing analysis and a correction factor can be applied to acquire estimates of the timing properties.

7.7 ECU use case "Verification of Timing"

In this use case the objective is that this system satisfies a given set of timing constraints, for example "from sensor to actuator".

7.7.1 Characteristic Information

Goal In Context:	Verify the timing of a defined system

Brief Description:	Verify the timing to ensure the functionality of the system and that all given timing constraints are fulfilled. The verification of the timing can be conducted via various timing analysis methodologies, e.g.: • response time analysis • schedulability analysis • runtime measurement comparison The selection of timing analysis method depends on the demanded level of accuracy and the type of timing constraint that should be verified. The timing model, describing the necessary timing behavior of a functionality, can vary as well depending on the system model granularity.
Scope:	ECU
Frequency:	Whenever the decision has been taken to verify the timing of the existing system. Exemplary triggers to start the timing verification can be: • adding, removing or modifying the SWE to ECU mapping • modification of the internal behavior of a SWE • reconfiguration of the system schedule e.g. changing process priorities • updating the bus communication, see for further information in chapter 6
Precondition:	The following preconditions must be fulfilled to execute the described use-case on the level of ECU: • The the software configuration of the ECU is valid and its description is available. • The SWE that are mapped to the ECU which is the subject of timing analysis are valid. • Definition of relevant timing constraints, which should be satisfied. • Timing model adapted to the granularity of the available system model.
Success End Condition:	Timing analysis indicates that the timing constraints are fulfilled in all system states. All relevant documentation has been updated.
Failed End Condition:	Neither of the applied timing analysis methodologies indicate that all timing constraint are satisfied. Timing measurement comparison indicates that at least one timing constraint is violated.
Actor(s):	ECU Integrator, Timing Engineer

Table 7.6: Characteristic Information of ECU UC "Verification of timing"

7.7.2 Main Scenario

A systematic approach for this use case is depicted in figure 7.7. The following steps typically apply:

1. The use-case begins with the decision to execute Task “Perform Model-Based Timing Analysis” or Task “Perform Implementation-Based Timing Analysis” for a specific system. This is usually done after modifying the SWE behavior or changing the system configuration.
2. The Timing Engineer of the UC conducts timing analysis as described in section 9.5
3. The Timing Engineerr executes the Task “Verify Timing” and concludes whether all timing constraints are fulfilled (e.g GENERIC PROPERTY Load, SPECIFIC PROPERTY Execution Time, GENERIC METHOD Determine Latency, Interrupt Load) or at least one is violated.
4. If all constraints are fulfilled, the ECU Integrator approves the work products as valid. Approved work products are:
   - the Timing Model and Timing Requirements Document (TIMEX extract) (see table Work Products)
   - the Timing Analysis Report (see table Work Products)

   In case at least one constraint is violated, the typical procedure is described in ECU use case “Debug Timing”.

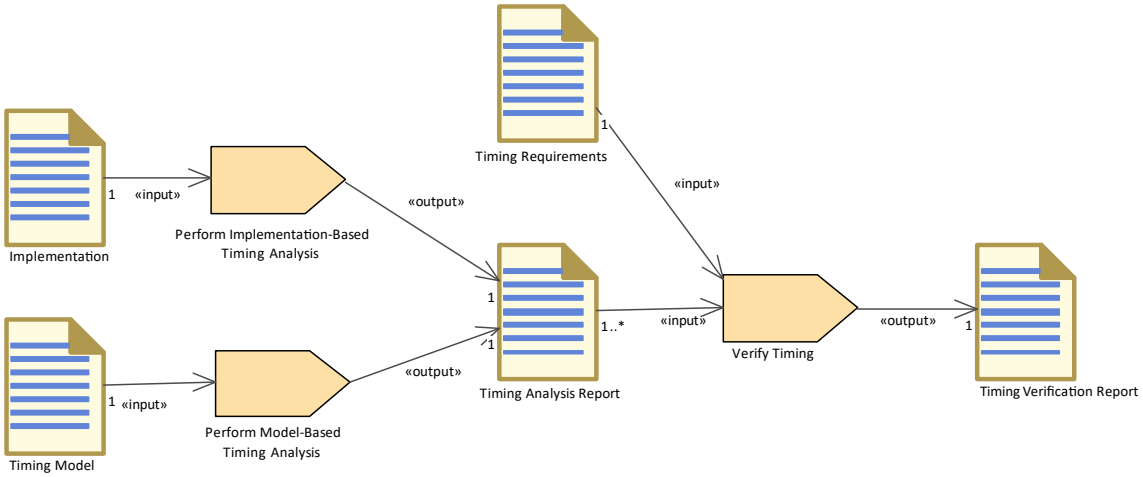

Figure 7.7: SPEM process model for ECU use-case “Verification of Timing”

7.8 ECU use case “Debug Timing”

Whenever an ECU shows sporadic system crashes, data inconsistencies or unexpected overload scenarios, delays or jitters, a timing issue could be the cause of the problem. Tracking the problem down with conventional debug methods can be very painful and time consuming. This is also true even if a certain problem is very obviously related to timing.
Before any problem can be *solved*, it has to be understood. This is what timing debugging is about: understanding a timing problem that is present on a real ECU. Once

the problem is understood, the solution finding and solving follows, see section 7.9 ECU use case “Optimize Timing of an ECU” on page 117. This use case focuses on debugging the timing of a single ECU, e.g. the ASA shown in figure 7.1 on page 103.

7.8.1 Characteristic Information

Goal In Context:	Understand a (timing) problem and isolate the cause of the problem.
Brief Description:	Using dedicated timing debugging methods (see chapter 9), debug a problem and find out, if it is a timing problem. If so, track down the cause of the problem so that it is completely understood. This makes solving the problem possible in a next step.
Scope:	ECU
Frequency:	Whenever a not trivial problem is detected in the ECU.
Precondition:	A running system
Success End Condition:	Problem understood, cause of the problem isolated. Artifacts: set of test conditions that can reproduce the problem, documentation describing the problem, e.g. schedule traces
Failed End Condition:	• problem not understood or • problem is not caused by faulty timing or • problem is not reproducible or based on the data of previous occurrences not sufficiently analyzable.
Actor(s):	Timing Engineer, ECU Integrator, Test Engineer

Table 7.7: Characteristic Information of ECU UC “Debug Timing”

7.8.2 Main Scenario

A systematic approach for this use case is depicted in figure 7.8. The following steps typically apply:

1. The use case begins when the ECU Integrator is confronted with a timing problem or a problem that directly affects the timing behavior on a real ECU (Task “Create Implementation”).
2. The Test Engineer sets up a test environment in which timing debugging can take place.
   (a) If the failure cause can be provoked in a reliable manner use the real system for timing debugging (step 4).
   (b) If it can’t an iterative approach is necessary:
   - Obtain as much information about the environment and circumstances of the timing problem from the original reporter, e.g. log files, telemetry, HW and SW data sheets.

- Build a test setup and define the set of test conditions based on this information.
- Run tests to provoke the failure cause. If this succeeds, continue with step 4.
- If the failure cause could not (yet) be provoked, get more information from the original reporter.
- Analyze and minimize the differences between your test setup and the real environment in which the timing problem occured.
- If time, money or other budgets for the analysis are depleted exit the use case and either report 'could not reproduce (CNR)' or continue with step 3 (not recommended).
- Otherwise continue with step 'Run tests to provoke the failure cause'.

3. If the iterative approach fails but CNR is not accepted by one or more stakeholders create theoretical failure models using the data of previous occurrences (Task "Perform Implementation-Based Timing Analysis") and techniques like Ishikawa diagrams (also called fishbone diagrams). This approach incurs a huge amount of work and little focus on the (as yet unknown) failure cause. It should be considered a last resort if CNR is not acceptable.
4. The Timing Engineer and ECU Integrator debug and analyze the timing behavior to identify the cause of the problem. Dedicated timing analysis methods (e.g. trace-based, see Task "Perform Implementation-Based Timing Analysis") and section 9.5) can be used for this purpose.
5. Isolate the problem.
6. In a next step, the problem can be fixed using the set of test conditions that can reproduce the failure (see ECU use case "Optimize Timing of an ECU").

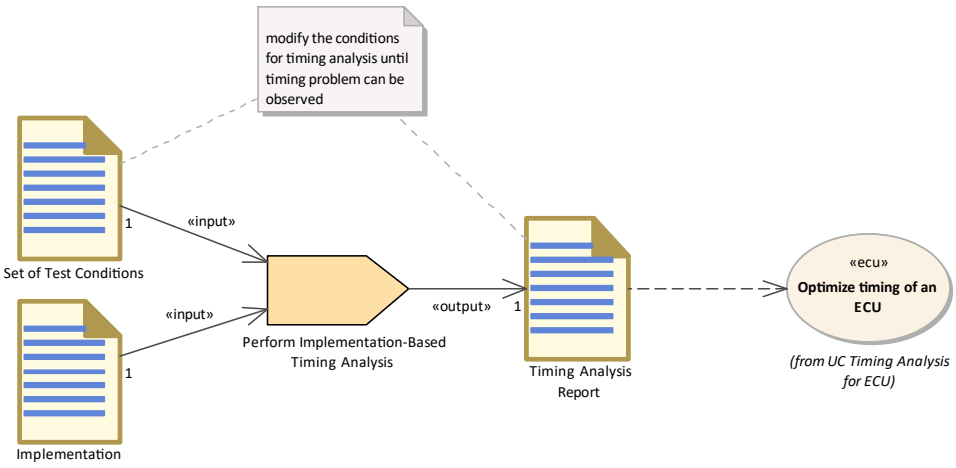

Figure 7.8: SPEM process model for ECU use case "Debug Timing"

7.9 ECU use case “Optimize Timing of an ECU”

The main idea behind this use-case is to optimize the timing behavior of a working ECU. Sometimes the resource consumption is higher than expected or it is required to integrate further SW-C into the ECU. Optimization is also requiered if timing problems have been identified and now need to be patched.

Different performance key indicators are possible:

- load balancing (distribute load on time axis, load balancing over different cores)
- minimize systematically response times, jitters etc.
- reduce number of preemptions (and thus reduce OS overhead)
- reduce number of migration (and thus reduce migration overhead)
- reduce resource consumption (inter-core communication, memory (buffer sizes), load)
- reduce number of scheduling interrupts
- reduce waiting times

See also chapter timing properties 9.

Sub-use-case(s): ECU use case “Optimize Scheduling” and ECU use case “Optimize Code”.

7.9.1 Characteristic Information

Goal In Context:	Remove timing violations (optimize resource consumption, data consistency, reduce jitter,..) or minimize resource consumption
Brief Description:	Based on timing requirements, while taking all timing constraints into account the overall timing architecture for an ECU is optimized
Scope:	ECU
Frequency:	Whenever a timing violation is detected in the ECU, an additional functionality is added/expected or existing functionality is modified
Precondition:	A running system and/or ideally a useful system description (timing-model)

Success End Condition:	Found a better solution which fulfills all timing and resource requirements (even with additional functionality if applicable). Artifacts: • New Schedule • Updated Timing model • Optimized code • New memory layout • New code generator options • New compiler options
Failed End Condition:	No solution found
Actor(s):	Timing Engineer, ECU Integrator

Table 7.8: Characteristic Information of ECU UC "Optimize Timing of an ECU"

7.9.2 Main Scenario

A systematic approach for this use case is depicted in figure 7.9. The following steps typically apply:

1. The use case begins when the ECU Integrator becomes aware of timing violations or the need to add more functionality into an already heavily loaded system. This can be conducted after executing Task "Perform Implementation-Based Timing Analysis" and Task "Verify Timing".
2. Analyze the current system (verify the timing of the system, see ECU use case "Debug Timing") and find hot-spots . These are situations in the schedule, where either timing requirements or resource consumption constraints are violated already or would be if more load was added.
3. Definition of the optimization goal(s) on a per hot-spot basis.
4. Analysis of available options in order to relax the hot-spots. These options can include modification of the scheduling configuration by ECU use case "Optimize Scheduling" and/or code optimization in ECU use case "Optimize Code". For each option, continue with the corresponding use case.
5. The ECU Integrator performs a trade-off analysis to weight the different possibilities for the optimization of the timing and its impact on the system
6. The ECU Integrator decides for a modification and changes the timing-model/the code of the system.
7. The Timing Engineer validates the timing of the ECU by doing Task "Verify Timing"
8. Verification against optimization goal

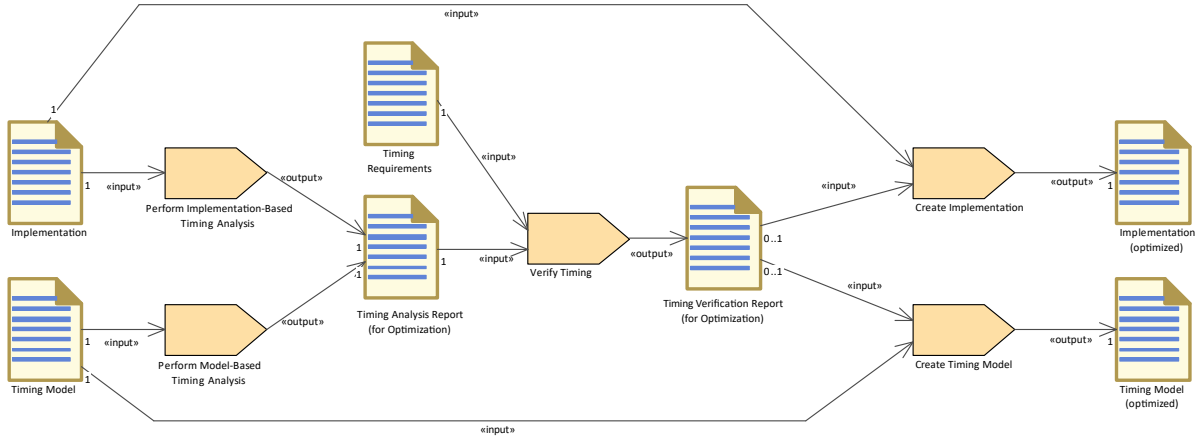

Figure 7.9: SPEM process model for ECU use-case “Optimize Timing of an ECU”

7.10 ECU use case “Optimize Scheduling”

The main idea behind this use case is the optimization of an existing schedule of a working ECU with a defined goal such as “remove local overload” or “reduce response time of task xyz”.

7.10.1 Characteristic Information

Goal In Context:	Fulfill predefined optimization goal
Brief Description:	Find a modified schedule configuration which fulfills the goal without causing new timing violations or violates resource constraints
Scope:	ECU
Frequency:	Whenever a timing violation is detected in the ECU, an additional functionality is added/expected or existing functionality is modified
Precondition:	A running system and/or ideally a useful system description (timing-model)
Success End Condition:	Found a modified schedule configuration which fulfills the goal without causing new timing violations. Artifacts: • New Schedule, better than the original schedules with respect to a specific timing properties, see chapter 9.4 • Updated Timing model
Failed End Condition:	No solution found
Actor(s):	Timing Engineer, ECU Integrator

Table 7.9: Characteristic Information of ECU UC “Optimize Scheduling”

7.10.2 Main Scenario

This use-case typically consists of the following steps:

1. The use-case begins when the ECU Integrator is confronted with a certain optimization goal regarding the scheduling
2. Analysis of available options. Depending on the platform different options to modify the scheduling are available:
   - For CP e.g. modification of the runnable to task mapping, the runnable sequence/order inside tasks, the allocation of task to different cores, the partitioning of tasks into smaller entities for load balancing, the change of priorities/offsets/recurrences of tasks
   - For AP e.g. the constrains in the Execution Manifest for a Thread can be relaxed to give the Scheduler more freedom on using the computational resources or add additional constrains to the Execution Manifest to guaranty computational resources for a Thread to achieve a desired response time. For multi-threaded Applications adjusting the scheduling may require to change its implementation.
3. The ECU Integrator performs a trade-off analysis to weight the different possibilities for the optimization of the schedule and its impact on the system (Task “Verify Timing”)
4. The ECU Integrator decides for a solution and modifies the timing-model/code of the system.
5. The Timing Engineer verifies the timing of the ECU by conducting response time analysis, scheduling analysis or measurements (Task “Perform Model-Based Timing Analysis” or Task “Perform Implementation-Based Timing Analysis”)
6. Verification against optimization goal (Task “Verify Timing”)

7.10.3 Scheduling and Sporadic Events

A challenge when optimizing the scheduling strategy is dealing with sporadic events. This is especially challenging for ECU that need to handle large numbers of sporadic event and also implement functions requiring real time behavior with high timing accuracy (e.g. a gateway that needs to process large numbers of reception event and transmission confirmation events from various networks and also implement a time synchronization master).

A the beginning the timing relevant properties of the sporadic events need to be analyzed. The following properties should be determined:

- Execution time of schedulable entities triggered by sporadic events
- Inter-Arrival Time of sporadic events (average and worst-case)
- Maximum number of sporadic events in certain time intervals. Since for periodic sampling the time interval is an optimization parameter, it is advantages to have

an analysis function that can determine the number of event for any given time interval.

In general interrupt service routines should have minimal execution time, to reduce the impact on the schedulers behavior. For functions that require a very high timing accuracy, it may be required to disable interrupts for its execution. The interrupt lock time should be very short and only be used for critical section.

Rare sporadic events can be handled by polling the event status from a periodic function, in case the response time requirement is not very strict. If a faster response time is required, the scheduler can be notified about the occurrence of the event in the interrupt service routine and then schedule schedulable entities for processing the event. In both cases processing of the event can be incorporated in the scheduling strategy and optimized by assigning priorities. Only if very fast reactions to events are required, should the reaction be implemented in the interrupt service routine. In this case it has to be verified, that the timings of the other schedulable entities are not invalidated by the occurrence of such an interrupt event.

A general approach to deal with bursts of sporadic events, is to buffer the events and process the events deferred in periodic schedulable entities. The timing behavior is impacted by the buffer size, buffering strategy and the processing period. It should be noted, that buffer size is also limited by the available memory resources, which need to be considered, but are outside the scope of this document. If only the latest event is relevant and it is acceptable to miss some events, a last-is-best buffering strategy can be used, which requires only a single buffer entry. If all events are important, the buffer needs to be able to hold all events that can occur during a processing period. Increasing the processing period reduces the CPU load, but increases duration of processing blocking schedulable entities with lower priority and increases the response time for processing the events. For low priority events it is possible to schedule the processing with low priority with and a long period, to minimize the impact on other schedulable entities.

Buffering of events also allows to implement more sophisticated priority schemes. Technical limitations cause interrupts to always be executed with a defined order. Even if the interrupts have the same priority configured, will the order be fixed, in which these interrupts are processed. This can result in long response times for interrupts, that come late in the processing order. By buffering the interrupt events, it is possible to implement other priority schemes like round robin, to achieve more equally distributed response times for different interrupt events.

7.11 ECU use case “Optimize Code”

Since the code and the deployment of code has a huge impact on timing, different optimization activities can be performed. The scope of the optimization can be different (memory, run-time, safety, re-usability, easy to understand, etc.), however in the scope of this document, the optimization scope is limited to timing effects. But it has to take into account, that such timing optimization influence other aspects of the system, such as memory or re-usability and that such optimization is constrained by safety or security aspects

7.11.1 Characteristic Information

Goal In Context:	Optimize the code with respect to timing. Typically: minimize the WCET, the average execution time or both.
Brief Description:	Based on timing requirements optimize the overall timing architecture for an ECU
Scope:	ECU
Frequency:	Whenever a timing optimization in the ECU is needed.
Precondition:	Code available (ideally compilable, linkable and executable on the target platform)
Success End Condition:	Found a better code with respect to timing. Possible artifacts: • Optimized code • New memory layout • New code generator options • New compiler options
Failed End Condition:	No solution found
Actor(s):	Software Component Developer, Timing Engineer

Table 7.10: Characteristic Information of ECU UC “Optimize Code”

7.11.2 Main Scenario

This use-case typically consists of the following steps:

1. The use case begins when the Software Component Developer determines to optimize a certain code fragment (a schedulable entity, a function or part of a function) usually after doing either Task “Perform Model-Based Timing Analysis” or Task “Perform Implementation-Based Timing Analysis”
2. Definition of optimization goals, e.g. reduction of core execution time or reduction of time spent time spend in an Exclusive Area.
3. Analysis of available options, e.g. different compiler options, code refactoring or implementing a different algorithm
4. Modification, pick at least one of the options and implement it

5. Verification of the functional behavior, e.g. run unit test
6. The Timing Engineer verifies the timing optimization goal by executing Task "Perform Implementation-Based Timing Analysis"

7.12 ECU use case "Integrate a new function"

The use case describes the integration of a new function on to an ECU, with a focus on the timing aspect of the integration. For a general description of the ECU integration use case refer to AUTOSAR Methodology [1] or Methodology for Adaptive Platform [17] respectively. A new function may be fully or partially implemented on an ECU and represented by one or more SWEs.

7.12.1 Characteristic Information

Goal In Context:	Integrate a new function on to an ECU while maintaining existing timing constrains and fulfilling timing constrains of the new function.
Brief Description:	A new function is integrate on to the ECU, while taking all timing constraints into account.
Scope:	ECU
Frequency:	Every time a new feature or vehicle function is added.
Precondition:	The current working implementation or timing model of the ECU as starting point. The SWE implementing the new function are available, with all input data required for SWE integration according to AUTOSAR Methodology [1] or Methodology for Adaptive Platform [17]. The ECU Timing (see table 10.12) information including the new function is available.
Success End Condition:	The new function is fully integrated and the timing requirements are satisfied.
Failed End Condition:	The new function cannot be integrated without violating at least one timing requirement.
Actor(s):	ECU Integrator, Timing Engineer

Table 7.11: Characteristic Information of ECU UC "Integrate a new function"

7.12.2 Main Scenario

A systematic approach for this use case is depicted in figure 7.10. This use-case typically consists of the following steps:

1. The ECU Integrator checks the required input for availability and completeness and gets a quick overview on the integration items.
2. Integrate implementation of new function:
   - For CP: The integration shall be performed as described in AUTOSAR Methodology [1] (e.g. configuration of BSW to fulfil SW-C service needs,

mapping SW-C to a partition according to ASIL classification). For the timing behavior the main focus during integration is definition of task and their scheduling and the mapping of the runnables to these tasks. For the runnable mapping the ECU Integrator shall consider the priority of time critical runnables, event chains between runnables, runtime of the tasks and grouping of runnables with the same period or periods where the shortest period is a common denominator for the other runnables.

- For AP: The integartion shall be performed as described in Methodology for Adaptive Platform [17] (e.g. perform communication and diagnostic mapping, if required by the new function, update the Execution Manifest to include the new Adaptive Executables or update Function Group definitions to include the new Function)

3. Analysis of the system with the new function and verification of the timing constraints. For this the Timing Engineer needs to perform the task Task “Perform Model-Based Timing Analysis” or Task “Perform Implementation-Based Timing Analysis” followed by Task “Verify Timing”

   (a) Analyze the load on the CPU core(s) of the ECU. For a specific scenario and time frame, the CPU load shall not exceed a predefined limit. The output of the analysis is the timing property GENERIC PROPERTY Load obtained with the timing method GENERIC METHOD Determine Load.

   (b) Analyze the latency for time critical event chains. The response times for event chains from an input event (e.g. sensor measurement or frame reception) to an output event (e.g. frame transmission or actuator control) GENERIC PROPERTY Latency are obtained with the timing method GENERIC METHOD Determine Latency. The values of the timing properties are compared to the defined requirements. It can be useful to compare the results to the previous timing properties. This can help to understand the impact of the changes and help with future decisions, if additional optimization steps are required.

   (c) Analyze the jitter of periodic schedulable entities requiring accurate cycle timing.

4. If the timing verification failed, multiple iterations on optimizing the scheduling can be performed.

   (a) From Timing Analysis Report (see table 10.12) hot spots can be identified and the optimization goals can be defined.

   (b) Based on the identified hot spots possible improvements to the scheduling can be derived.

   (c) The ECU Integrator performs a trade-off analysis , to weight the different possibilities for the optimization of the timing and its impact on the ECU.

   (d) The Timing Engineer then verifies if the ECU meets all timing requirements after updating the scheduling.

5. If it is not possible to create a scheduling that fulfils the timing requirements, an optimization of the code by the Software Component Developer or remapping of the SWEs by the Function Architect can provide a solution. Afterwards a new integration attempt can be started by the ECU Integrator.

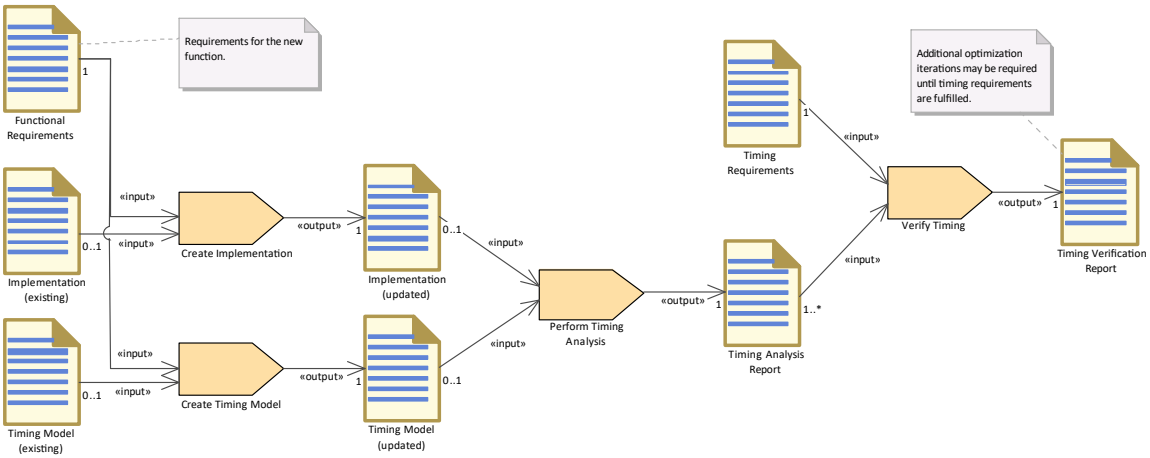

Figure 7.10: SPEM process model for ECU use-case “Integrate a new function”

8 System Level Logical Execution Time

8.1 Basic concepts of System Level Logical Execution Time

The LET paradigm can be used to abstract physical execution and communication as long as there is a common time base and all periods of a cause-effect chain are less than the corresponding LET intervals, $LET \leq period$. The AUTOSAR Timing Extensions for the classic platform provide specification means for LET intervals, where a LET interval references a group of executable entities. (Also called *runnable entities*, or short *runnables*, for the implementation of software components.) LET in AUTOSAR therefore provides the ability to specify an abstraction of computation and is not able to express communication latencies. For the important class of cases where one of the two assumptions does not hold, the LET paradigm has been extended to a *System Level Logical Execution Time (SL-LET)*. In general, this extension allows to cover two main problems in the design process, without altering the properties of programming with LET.

First, SL-LET intervals provide a logical timing abstraction for executable entities that are implemented on different clocks and/or are distributed. As an example, executable entities of a cause effect chain may be deployed to different ECUs with significant communication delays as well as different clock sources. Beside the implementation, SL-LET Intervals can also be used for specification means in earlier design stages, such as for specifying a deterministic data flow among coarse grained functional blocks. This includes the possibility to decompose and refine SL-LET intervals in the process of design concretization, unifying the timing abstraction from the functional level down to the software integration on ECU level. In the following, necessary requirements are formulated.

The assumption of instantaneous LET events for read and write actions implies that those actions take place in zero time. A correct implementation of LET, which is used for data exchange between executable entities deployed in the same ECU, must preserve the data flow semantics of LET. Note that within one ECU, executable entities can exchange data via shared memory and can execute on different cores.

In contrast to that, a distributed cause-effect chain, e.g. including two ECUs and a bus/network, raises two major challenges:

- First, LET events imply a *common understanding of time* that has to be enabled on the different computation resources and
- second, we cannot keep the requirement that the latency must not exceed the period. It is likely that the distributed communication via bus/network involves *significant communication latencies* and a timing model utilizing LET has to cope with large latencies.

Therefore, SL-LET provides three major extensions that are necessary to also apply the LET approach to larger cause-effect chains and distributed systems.

Extension 1) Permitted Pipelining Property (PPP): Without SL-LET, the upper bound for an LET interval ($LET \leq period$) was necessary to ensure that no schedulable entity is re-executed before an ongoing execution of that entity has finished and has written all its data. This is no specific requirement of a system using LET, but is a typical assumption in periodic real-time systems. If this bound is exceeded, overlapping executions of a schedulable entity (i.e. overlapping jobs) are possible. This is like re-executing a finite state machine before it has created its output. The result is generally incorrect, implementation dependent, and possibly violates the assumption of data-flow determinism. The cause for incorrectness is the access to data that are written in an unfinished preceding LET interval. If that cause can be removed, overlapping execution is possible. The term "pipelined" is borrowed from computer and software technology to enhance the concept of SL-LET with LET intervals that have duration greater than the period.

The *Permitted Pipelining Property (PPP)* is a property of a schedulable entity, stating that pipelined execution of schedulable entities in an LET interval with $LET > period$ is permitted. That holds if no data from schedulable entities in an unfinished preceding LET interval is accessed. The PPP enables a possible overlapping of executions in case of $LET > period$, thereby ensuring a deterministic data flow. This is shown in Appendix C.2 including an example. The definition is chosen, such that it not only extends classical LET as introduced in Section 2.3.1, but is a generalization of classical LET scheduling. The PPP extension allows to capture systems with $LET > period$. Since pipelining alone is not sufficient for distributed systems, also extensions 2 and 3 are necessary.

Extension 2) Logical Execution Time Zones (LET Zones): To support the meaning of LET events in a distributed system, a common understanding of time and timing events has to be enabled. This can be done by providing a sufficient time synchronization of the different local clocks (e.g. on different ECUs). One example approach is to use a clock synchronization protocol like PTP, to synchronize the local clocks to a master clock. An LET zone in SL-LET is a subsystem with a local time base that has a bounded clock deviation to other LET zones. Within a LET zone, the assumption of zero-time communication can be fulfilled while communication between time zones has a non-negligible but bounded delay.

Extension 3) Interconnect LET: Communication between two LET zones can be abstracted with an interconnect LET[1].

An interconnect LET is a specialized LET interval where data is read at an LET event in one LET zone and written at an LET event in another LET zone. As a consequence, the interconnect LET does not solely represent an execution/computation latency (e.g., within the COM stack), but also comprises a transmission/communication latency.

An interconnect LET may consist of a sequence (i.e. a unique chain) of schedulable entities. To enable communication times larger than communication periods, interconnect LETs are subject to the following two conditions:

[1] Also referred as *interconnect task* in [6]

- If the interconnect LET consists of a sequence of schedulable entities, this sequence must have a deterministic data flow.
- Two executions of an interconnect LET must be functionally independent in the sense that no schedulable entity of an interconnect LET must access data from an unfinished earlier execution. This requirement follows the same argument as extension 1 and is elaborated in Appendix C.2.

The first property is needed to make the interconnect LET compatible to the LET paradigm, the second condition naturally holds for communication tasks, where data are transported as net payload, rather than being modified. Together, the conditions are sufficient to permit $LET > period$ for interconnect LETs.

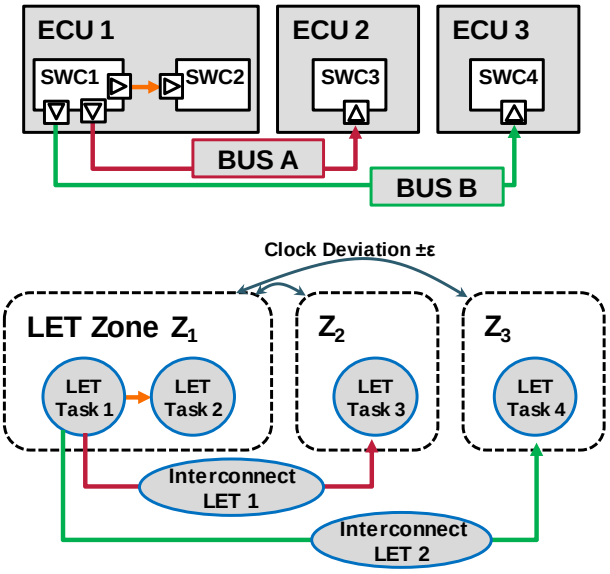

Figure 8.1: SL-LET model: LET Zones and Interconnect LETs

Figure 8.1 shows an example for a given mapping of software components to ECUs. The local communication on ECU1 can be specified with LET (by means of the Timing Extensions) and can be implemented, e.g., with shared memory communication. A *LET task*, as denoted in Figure 8.1, represents a group of executable entities (implementing the software components) which are assigned to a common LET interval. SL-LET enables to use the LET abstraction for remote communication to ECU2 and ECU3. Comparable to the abstraction of computation time with local LET, SL-LET provides an abstraction for communication times by the means of interconnect LETs. This is not restricted to the transmission time on a bus or network but allows also to cover the required basic software that is involved. As with LET, the SL-LET specification makes no assumption about the underlying network or bus technology as well as no assumption about completeness, meaning that it can be combined with non-LET tasks in a cause-effect chain.

Again, the extensions made by SL-LET are not restricted to inter-ECU communication. This is only used as an illustrative example here. The use-cases in the following sections provide examples

- Section 8.2: how to specify SL-LET for function timings, derive budgets for function networks and abstract event-driven cause-effect chains with periodic activation
- Section 8.3: how to specify a deterministic data flow with LET and SL-LET for SWCs without a given mapping.
- Section 8.4: how to decompose a SL-LET specification to an LET specification.
- Section 8.5: how to compose a SL-LET specification from an existing LET specification.
- Section 8.6: how to ensure robustness against timing changes.
- Section 8.7: how to decompose the system in LET zones.

Implications of an LET interval that is specified larger than the period on the implementation are discussed in Appendix C.2.

8.2 SL-LET in early design stages: Specification and budgeting on functional level

Use-case 4.4 has already motivated the partitioning of a vehicle function into a functional architecture (consisting of functional blocks and their interfaces). Decomposing the timing requirements in a consistent way is a non-trivial task as soon as a deterministic data-flow shall be enforced. As discussed in Section 3.1, SL-LET introduces a composable timing of functional blocks. The timing requirements identified in use-case 4.3 can be decomposed in smaller timing budgets which are associated to functional blocks as shown in use-case 4.4. The decomposed timing budget can be applied as the SL-LET interval for a functional block. The SL-LET semantics constrain a later implementation in a way that there will not be a propagation of execution time jitter across the borders of functional blocks.

The benefits can be discussed based on an abstract functional model of a sensor fusion and perception pipeline, which is shown in figure 2.3. It comprises

- multiple event driven cause-effect chains with periodic activation,
- dependent cause-effect chains including fork and join points,
- cyclic dependencies in cause-effect chains

Section 2.3 already outlined the challenges regarding data-flow determinism and data-age dispersion, when for example a sensor fusion function comprises dependent cause-effect chains. Only specifying time budgets as upper latency bounds does not guarantee a deterministic data flow in this scenario, since jitter propagates independently in each pipeline path. As a result, the relative data age of the different inputs that shall be combined varies. This data-age dispersion becomes problematic e.g.,

when one path observes a worst-case latency while the other one observes a best-case latency. As soon as the cause-effect chains comprise loops, the problem is amplified. Combining an output data sample, which originates in an uncertain data-age dispersion, again with new input data of uncertain data-age dispersion accumulates the non-determinism.

One option would be to annotate each data sample with a timestamp. This raises the question how timestamps are processed, propagated and inherited in the pipeline. First, processing timestamps depends on the timestamp granularity. It requires comparison of the timestamps to select samples from the input queues that shall be combined. As an example, even when the two cameras are triggered synchronously, their timestamps may slightly differ and a granularity has to be specified when two samples have the “same” time of origin. If multiple inputs with different sampling periods shall be combined, this becomes more complex. Next, the question remains how those timestamps are propagated resp. inherited when multiple inputs are combined, e.g., what timestamp is annotated to the outputs of the perception fusion in Figure 2.3. To ensure composability for later software updates, the existing timestamp information has to be preserved, which means that an output sample is annotated with all timestamps of the input samples it consists of. On the other hand, this impacts implementation, since the timestamp interpretation is tightly coupled to the function implementation.

Specifying SL-LET intervals for the different pipeline paths, as shown in Figure 8.2, allows to decouple timing aspects from the function implementation and to constrain a deterministic data flow. It enforces a fixed relative data age when multiple inputs (e.g., by the perception fusion) are combined and intercepts jitter propagation. Note that Figure 8.2 only provides a few SL-LET intervals for illustration. Implementing the SL-LET specification of one pipeline path is separated from its function implementation. As an example, a different Lidar classifier can be developed in isolation and can replace the existing Lidar pipeline. The SL-LET implementation is then responsible for providing the different processing jobs with their corresponding data samples and monitoring their adherence to the timing specification.

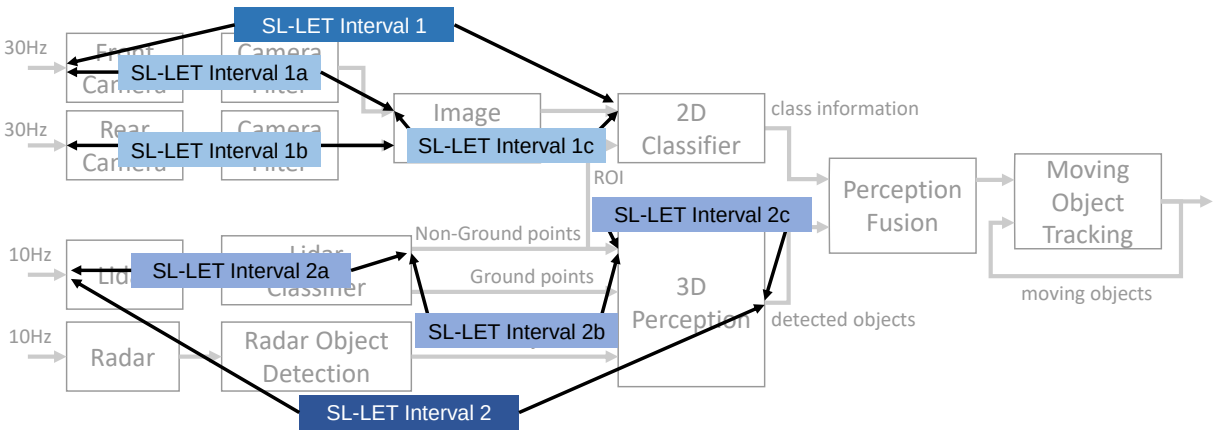

Figure 8.2: Exemplary SL-LET specification of event-driven processing pipelines including different types of decomposition

Applying SL-LET already on the functional level makes no assumptions on the granularity. As shown in section 4.1, a generic functional architecture comprises the required model elements

- **FunctionComponentType** (a functional block),
- **PortPrototype** (the input and output ports of a functional block) and
- **FunctionConnector** (connecting an output port of one functional block to an input port of another functional block).

SL-LET intervals can be formulated as timing constraints ranging between ports in form of

- a **Function response time budget** (for a single functional block),
- a **Function connector communication time budget** (for the connection between two functional blocks), or
- an **End-to-end time budget** for a coarse grained (mixed) specification.

In addition, a **Paradigm** has to be specified for each budget, to distinguish if it is an SL-LET interval or not, and optionally the **PermittedPipelining** has to be annotated, to allow for SL-LET intervals larger than their period. Table 8.1 shows the relation between the functional timing concept elements and TIMEX constraints for SL-LET.

Functional Timing Concept	*TIMEX Constraint*
Paradigm	`TimingDescriptionEventChain.category=SL_LET_INTERVAL`
PermittedPipelining	`TimingDescriptionEventChain.isPipeliningPermitted`
Function response time budget	`LatencyTimingConstraint.maximum`
Function connector communication time budget	
End-to-end time budget	

Table 8.1: Transformation from Functional Timing Concept to TIMEX Constraints for SL-LET

An overall time budget such as “SL-LET Interval 2” may be de-composed in smaller budgets for

- a set of connected functional blocks representing a portion of the processing pipeline (“SL-LET Interval 2a”),
- a specific functional block (“SL-LET Interval 2c”), or
- connections between two functional blocks (“SL-LET Interval 2b”)

The first case already shows how data-flow determinism can be specified for event driven processing pipelines that have an initial periodic activation. Further examples regarding the decomposition of SL-LET intervals as well as pipelining are discussed in the following sections.

8.3 Specify a deterministic data flow by using SL-LET intervals for SWC without given mapping

LET and SL-LET allow to specify a deterministic data flow among cause-effect chains by prohibiting, resp. masking, implementation specific execution and communication time jitter. This becomes of particular importance since cause-effect chains typically comprise fork and join points as shown in Figure 8.3.

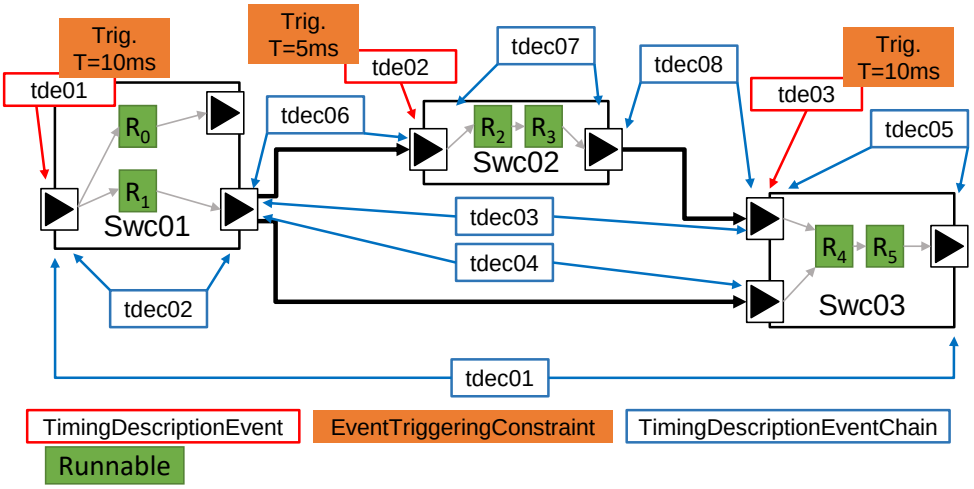

Figure 8.3: Cause effect chains comprising fork and join points, annotated by means of the AUTOSAR TIMEX

8.3.1 Problem: Data-age dispersion without LET and SL-LET

The example in Figure 8.3 includes `TimingDescriptionEvent`s for the activation of SWCs. By specifying `EventTriggeringConstraint`s, the three SWCs result in a periodic system with two sampling rates, 10ms and 5ms. `TimingDescription-EventChain`s (*tdec01* to *tdec08*) are included to relate read and write events on the ports of the SWCs to each other. To each `TimingDescriptionEventChain`, a `LatencyTimingConstraint` can be annotated, e.g., specifying the minimum and maximum latency of the chain.

If only upper latency bounds instead of SL-LET intervals are specified, the relative data ages at the input of `Swc03` can only be predicted to stay within a given interval (called the data-age dispersion). This is due to the join point at the input of `Swc03`, where one cause-effect chain may observe the best case (shortest) latency, while the other one observes the worst case (longest) latency. The resulting challenges have already been outlined in section 2.3.

8.3.2 Specifying Deterministic data-age dispersion with SL-LET

LET intervals as well as SL-LET intervals can be specified by means of the AUTOSAR Timing Extensions. There is one important conceptual difference between the specification of LET and SL-LET intervals. SL-LET intervals describe the timing of a data flow, wherefore they refer to input and output ports of functional blocks or software components. LET intervals on the other hand provide a skeleton for the execution times of executable entities, wherefore a LET interval in AUTOSAR references a group of executable entities. This use-case focuses on the hardware agnostic specification of SL-LET intervals while the decomposition from SL-LET to LET intervals for a given hardware mapping is discussed in Section 8.4.

A SL-LET interval can be specified by adding a `TimingDescriptionEventChain` with the `category=SL_LET_INTERVAL` as indicated in Figure 8.4. The `TimingDescriptionEventChain` references the release and terminate event of the SL-LET interval, both of type `TDEventSLLET`. More specifically, the subclass `TDEventSLLETPort` allows to reference the SWC ports. The SL-LET interval length is expressed with a `LatencyTimingConstraint`, while a `PeriodicEventTriggering` constraint specifies the beginning of the SL-LET interval (optionally with an offset). Table 8.2 lists the parameters period, interval length, and the offset related to a global hyperperiod for each SL-LET interval shown in Figure 8.4. It further references the corresponding `TimingDescriptionEventChain`s from Figure 8.3 which are omitted for readability in Figure 8.4. Further details regarding the Timex specification of SL-LET are described in [2].

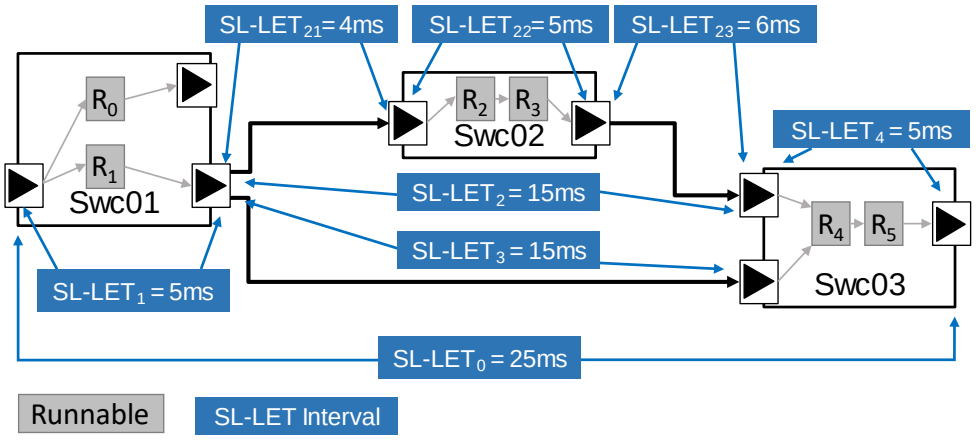

Figure 8.4: Cause effect chains comprising fork and join points including SL-LET intervals

SL-LET Interval	Corresponding Chain	Period	Interval Length	Offset
$SL\text{-}LET_0$	*tdec01*	10ms	25ms	0ms
$SL\text{-}LET_1$	*tdec02*	10ms	5ms	0ms
$SL\text{-}LET_2$	*tdec03*	10ms	15ms	5ms
$SL\text{-}LET_3$	*tdec04*	10ms	15ms	5ms
$SL\text{-}LET_4$	*tdec05*	10ms	5ms	20ms
$SL\text{-}LET_{21}$	*tdec06*	10ms	4ms	5ms
$SL\text{-}LET_{22}$	*tdec07*	5ms	5ms	9ms
$SL\text{-}LET_{23}$	*tdec08*	5ms	6ms	14ms

Table 8.2: parameters of the SL-LET intervals in Figure

As a consequence, an end-to-end deadline (e.g., a maximum latency associated with *tdec01*) can be partitioned in multiple SL-LET intervals. Multiple parallel cause-effect chains that lead to a join point may even be specified with an equal SL-LET interval (such as *tdec03* and *tdec04*), enforcing a deterministic and pre-defined data-age dispersion at the input ports of `Swc03`. SL-LET intervals can be further de-composed in smaller SL-LET intervals. In this example (see Figure 8.4), SL-LET_0 is a composition of SL-LET_{1-4} and SL-LET_2 is further decomposed into SL-LET_{21-23}. This is important in the design process as it allows to specify time budgets on the highest hierarchy level (in this example *tdec01* resp. SL-LET_0) and then concretize them during later phases of development. A decomposition of SL-LET intervals therefore corresponds to the decomposition of event-chains.

It also does not matter if a SL-LET interval comprises computation or not, since SL-LET intervals are used on here to specify the timing of the data-flow between ports of software components. This covers both, an input-to-output-port relation (where a SwC and therefore computation is involved) as well as an output-to-input-port relation (where only abstract communication is specified). For the latter case, communication between SwCs can be specified in an early phase of the development to have a time budget available during implementation. In Figure 8.4, $\text{SL-LET}_{2,21,23,3}$ are examples for such communication budgets.

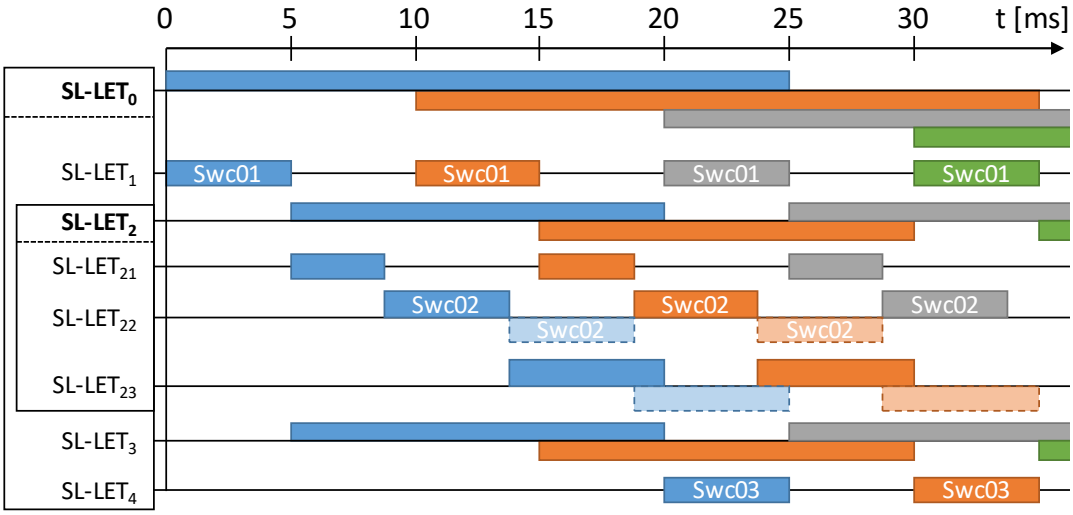

Figure 8.5: Representation of SL-LET intervals including their hierarchy (left boxes) and the resulting data flow (colored boxes)

Figure 8.5 provides a graphical representation of the data flow that is the result of the SL-LET intervals and their offsets. The colors blue, orange, grey and green are used to differentiate the first four executions of the cause-effect chain. All SL-LET intervals can therefore be associated with exactly one color. The partitioning of SL-LET intervals is denoted at the left side.

tdec01, *tdec03*, *tedec04*, and *tdec08* in Figure 8.4 already provide examples where the SL-LET interval is larger than the input period, which can be easily identified by overlapping instances of SL-LET intervals in Figure 8.5. As further discussed in Appendix C.2, this specification is allowed if the *Permitted Pipelining Property (PPP)* (ref. Section 8.1) is met, e.g.:

- **tdec04:** The SL-LET interval abstracts a communication that has no internal state (ref. interconnect LET in SL-LET, Section 8.1).
- **tdec03:** If the SL-LET interval comprises computation which is potentially stateful (such as `Swc02`), it has to be partitioned in smaller intervals (*tdec06-tdec08*).

8.3.3 Subsequent use-cases

This use-case does not incorporate any mapping of SWCs and communication to hardware yet. Instead it acts as a baseline for further use-cases:

- Section 8.4: Decomposition of SL-LET intervals to LET intervals for a given hardware mapping.
- Section 8.5: Composition of SL-LET intervals from an existing LET specification.
- Section 8.6: Robustness of cause-effect chains against modified execution and communication times with LET. Ensuring composability in case of changing network latencies with SL-LET.

8.4 Decomposition of SL-LET intervals to LET intervals

In an early stage in the development process, exact timings of SWCs are not available due to the lack of concrete implementations of executable entities. Decomposing a SL-LET interval to LET intervals allows to derive timing requirements for executable entities or groups of executable entities (more precisely, for their implementation). In such a way, fulfilling the derived LET intervals ensures compliance with the overall SL-LET interval and thus meeting the superordinate timing requirements of the cause-effect chain.

This use-case combines the SL-LET specification from the use-case in Section 8.3 with an exemplary hardware mapping, which is shown in Figure 8.6. The executable entities (more precisely the runnable entities) from `Swc01` and `Swc02` are executed on `ECU1` while `Swc03` is implemented on `ECU2`. As a consequence, the SL-LET intervals SL-LET_{23} and SL-LET_{3} refer to distributed communication, e.g. via a bus or network between both ECUs, while the SL-LET interval SL-LET_{21} refers to intra-ECU communication. Based on the initial SL-LET specification and the given Swc mapping, SL-LET intervals for computation can be decomposed to LET intervals. In contrast to SL-LET intervals, LET intervals reference a group of executable entities (runnables). The decomposition of SL-LET intervals to LET intervals again follows the needs of the

development process. In an early stage in the development process, the VFB is used to specify communication relations between SwCs. At this point, there is no definition of the implementation of executable entities yet, wherefore the SL-LET intervals are used here. Later in the development, during SWC development phase, runnables are defined, and it is as well known which runnables read which inputs and provide which outputs. Therefore, in this phase, the SL-LET intervals can be decomposed in concrete LET intervals, such that the initial event-chains and dataflow (or data ages) are guaranteed.

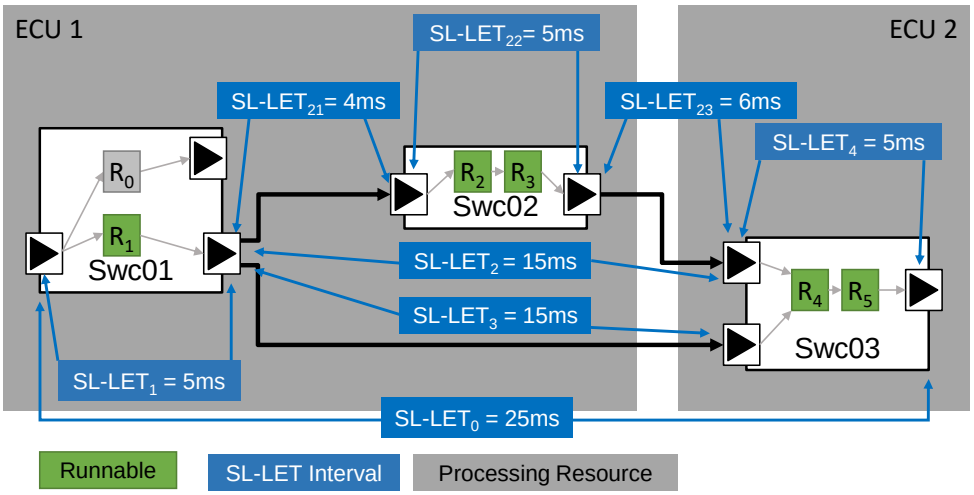

Figure 8.6: Mapping of SWCs to processing resources with SL-LET specification

The decomposition from SL-LET intervals to LET intervals is indicated in Figure 8.7. Just as with SL-LET intervals, LET intervals can be specified by adding a `TimingDescriptionEventChain` of the category `LET_INTERVAL`. Likewise the LET interval length is expressed with a `LatencyTimingConstraint` and a `PeriodicEventTriggering` constraint is used to specify the periodicity of the LET interval. Offsets between the release points of two LET intervals can be described by specifying an `OffsetTimingConstraint`. Other decompositions than displayed in Figure 8.7 are possible, as long as timing requirements and dataflow are fulfilled. Runnables from the same software component may use inter-runnable communication, wherefore the LET interval LET_2 specifies the input/output behavior of the group of runnables R_2 and R_3, which corresponds to the input/output behavior of `Swc02`. Table 8.2 lists the LET interval parameters for Figure 8.7, while the offsets are given with respect to the release point of LET_1. Figure 8.8 shows an example how an implementation may look like. As already described at Figure 8.4, $\mathrm{SL\text{-}LET}_2$ is a composition of $\mathrm{SL\text{-}LET}_{21}$, $\mathrm{SL\text{-}LET}_{22}$ and $\mathrm{SL\text{-}LET}_{23}$. $\mathrm{SL\text{-}LET}_0$ is composed of $\mathrm{SL\text{-}LET}_{1-4}$.

The resulting LET specification has to provide the same data flow that is demanded by the superordinate SL-LET specification. A straight forward approach is to derive LET intervals that match the properties of the corresponding SL-LET intervals. This is for example the case with LET_2 and $\mathrm{SL\text{-}LET}_{22}$. On the other hand, the specification of LET_1 also provides the same data flow for those two cause-effect chains as it is demanded by $\mathrm{SL\text{-}LET}_1$, although the interval length of LET_1 is shorter. As an example, the output of LET_1 may be used in a third cause-effect chain directly after the end of

the 3ms LET interval. This is always subject to the specific setup and the involved cause-effect chains.

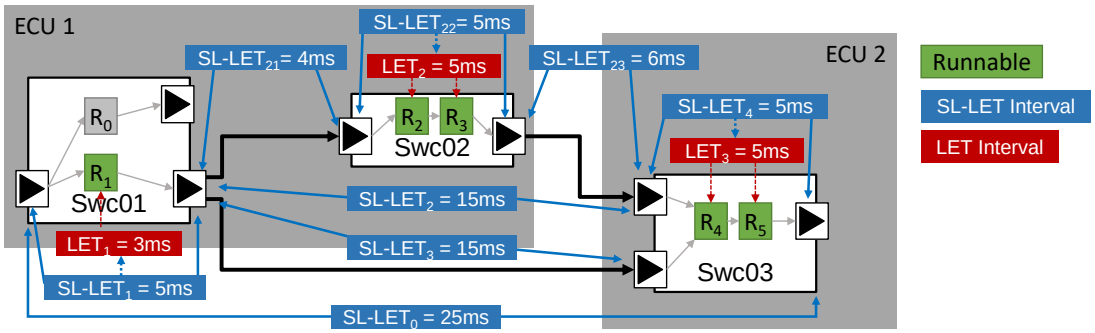

Figure 8.7: Decomposition of SL-LET intervals to LET intervals

LET Int.	SL-LET Int.	Runnables	Period	Interval Length	Offset to LET_1
LET_1	$\mathrm{SL\text{-}LET}_1$	R_1	10ms	3ms	—
LET_2	$\mathrm{SL\text{-}LET}_{22}$	R_2, R_3	5ms	5ms	9ms
LET_3	$\mathrm{SL\text{-}LET}_4$	R_4, R_5	10ms	5ms	20ms

Table 8.3: parameters of the LET intervals in Figure

For the inter-ECU communication, the example in Figure 8.8 assumes a time-triggered bus, where the messages between `Swc02` and `Swc03`, as well as between `Swc01` and `Swc03`, can be mapped to specific timeslots. The SL-LET specification ensures that the data is published for `Swc03` irrespectively of the chosen time slot at the end of the corresponding SL-LET intervals, here $\mathrm{SL\text{-}LET}_2$ and $\mathrm{SL\text{-}LET}_3$. On the other hand, the SL-LET interval $\mathrm{SL\text{-}LET}_{21}$ also remains valid in case of non-distributed communication (intra-ECU and based on the same clock), as the output of `Swc01` has to be buffered before `Swc02` may read it. This can be seen in Figure 8.8, where the offset of the LET interval LET_2 is a result of the decomposition from SL-LET to LET intervals. Although this is an intra-ECU delay for the given mapping, the SL-LET specification provides a degree of robustness for the case that `Swc02` is mapped to a third ECU.

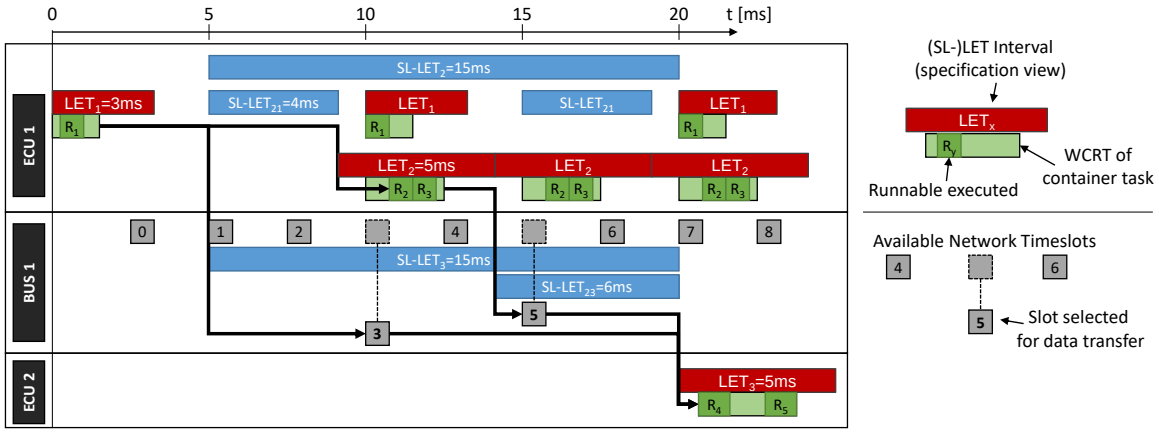

Figure 8.8: Decomposition of SL-LET intervals to LET intervals

8.5 Composition of SL-LET intervals from existing LET specification

This use-case describes how a SL-LET specification can be derived for a system that already has a LET specification, and under which constraints this is possible. LET intervals demand zero-time communication and therefore do not cover the communication of distributed SWCs, where communication takes a significant amount of time. A SL-LET interval can be composed of LET intervals to achieve independence of communication ideally, thus ensuring a deterministic timing of the data flow among distributed SWCs.

Figure 8.9 shows an example of three software components and their runnables, which are mapped to LET intervals. A LET interval in AUTOSAR specifies, that the associated executable entities are executed within the boundaries of the LET interval. This allows two different interpretations of the existing system, which are shown on the top of Figure 8.9.

Figure 8.9, left side: LET was solely used to specify the execution boundaries of a executable entities or a group of executable entities. Runnables read/write outputs according the LET semantics and the timing of the LET interval they are mapped to. The communication in-between has not been covered by the LET specification and there may be implicit assumptions resp. additional timing constraints. Examples are the zero-time communication assumption for shared memory on the local ECU or there exists a dedicated latency constraint for the remote communication. By reducing the focus of LET on the pure execution of executable entities, the initial specification does not guarantee a deterministic data flow without additional constraints for the communication. SL-LET can be used to close this gap and extend the specification, by explicitly modeling interconnect SL-LET intervals. A SL-LET interval for each software component can be directly derived from the existing LET intervals. The black arrows between the software components may lead to different types of communication during the design process, e.g. depending on the ECU mapping and the RTE generation. Therefore it is affordable to provide a dedicated SL-LET specification for those communication dependencies.

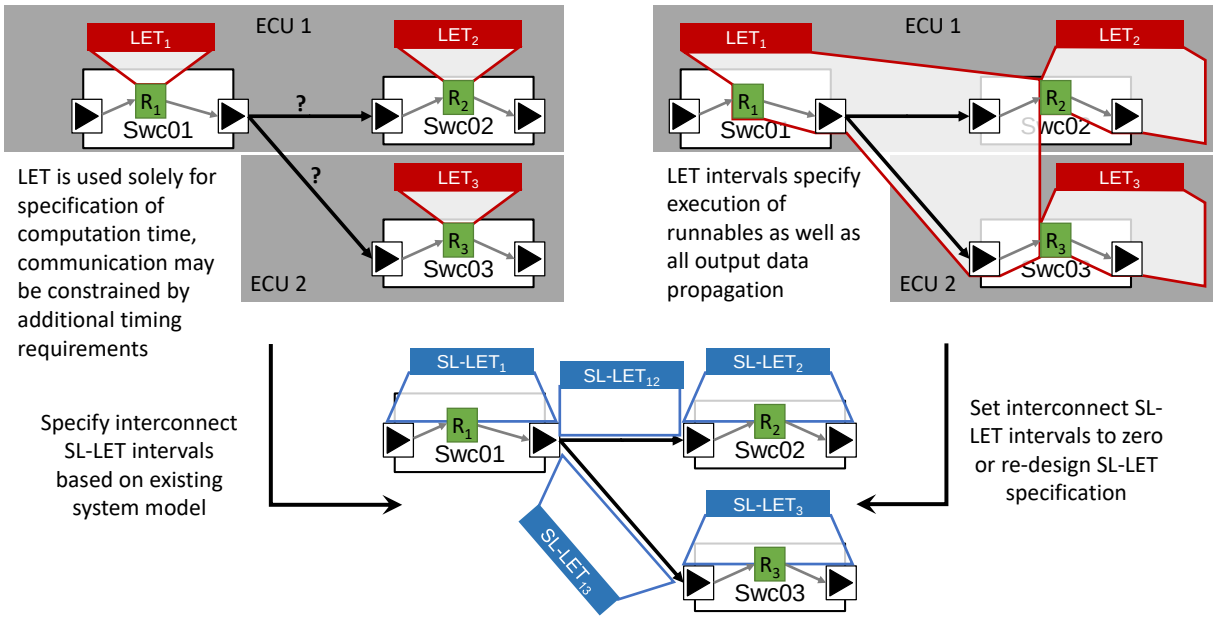

Figure 8.9: Composition of a SL-LET specification from an existing LET specification

Figure 8.9, **right side**: On the right side, a different view on a LET specification is denoted. It uses the explicit assumption that any form of data propagation to all subscribers is part of the publishers LET interval. Such a broadcast semantic is already known from synchronous systems but it has the drawback, that the LET interval comprises both, computation **and** communication. This specification provides a deterministic data flow but is very restrictive in terms of possible requirements on communication latencies. It requires a tight coupling of the runnable integration and the communication integration, since both share the same latency budget provided by the LET interval. However, this is can be handled in two ways, either by specifying the interconnect LET intervals to zero, which is very restrictive, or by re-designing the LET intervals. The first approach works for local communication that can already be implemented by shared memory, zero-time communication. A more efficient way is to create a SL-LET specification from the existing design (e.g., existing time budgets used for execution and communication), and then de-compose it to a new LET specification as discussed in use-case 8.4.

8.6 Robustness of cause-effect chains against modified execution and communication times with (SL-)LET

(SL-)LET provides a logical timing abstraction for computation as well as for communication. Based on an initial timing specification such as discussed in Section 8.3, the system may evolve and the timing behavior of an implementation may change. There are different sources for such changes, while prominent examples are:

- Software updates that lead to modified response times in a cause-effect chain due to a modified scheduling, although the updates belong to another cause-effect chain.

- Additional or modified network load that increases latencies for communication packets or even leads to a complete new network schedule in case of time triggered networks.
- Design modifications affecting the cause-effect chain such as a modified hardware platform or a new decomposition of timing budgets.

This use-case discusses the comparison of new timing requirements or a change of the timing behavior (e.g., due to a modified hardware platform) to the intended timing behavior as specified in the SL-LET interval.

8.6.1 Main Scenario

Figure 8.10 provides the SL-LET specification for the use-case as well as the exemplary hardware mapping known from Section 8.4. The runnable entities from `Swc01` and `Swc02` are executed on ECU1 while `Swc03` is implemented on ECU2. This use-case focuses only on the timing up to the input of `Swc03`, more precisely on the adherence to the specification of the SL-LET intervals SL-LET_1, SL-LET_2 and SL-LET_3.

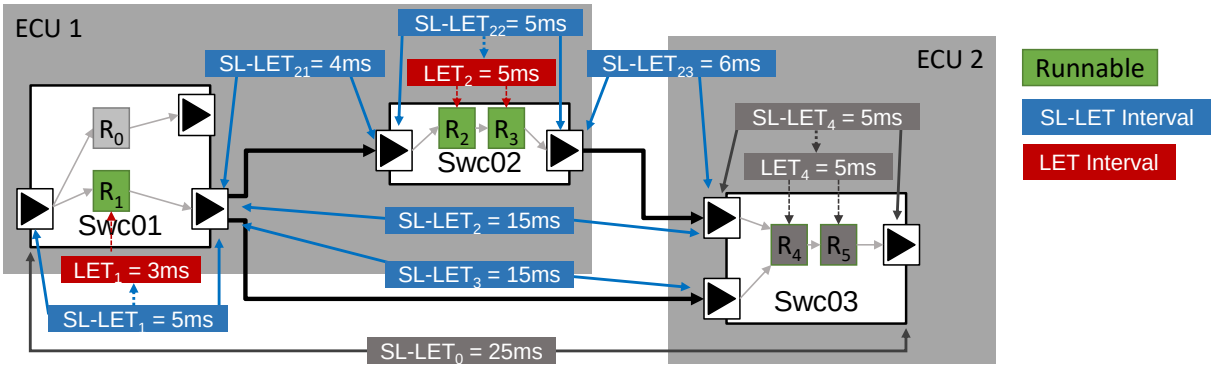

Figure 8.10: SL-LET specification combined with ECU Mapping and decomposed LET intervals

SL-LET allows to explicitly identify the robustness of the data flow to a modified timing behavior. Figure 8.11 shows a combined timing diagram of specified SL-LET intervals (blue boxes) and LET intervals (red boxes) as well an exemplary schedule of runnables (green boxes) and network packets (grey boxes). For a container task, the *robustness margin* is the difference between the worst-case response time of the task and end of the corresponding (SL-)LET interval and is highlighted in yellow.

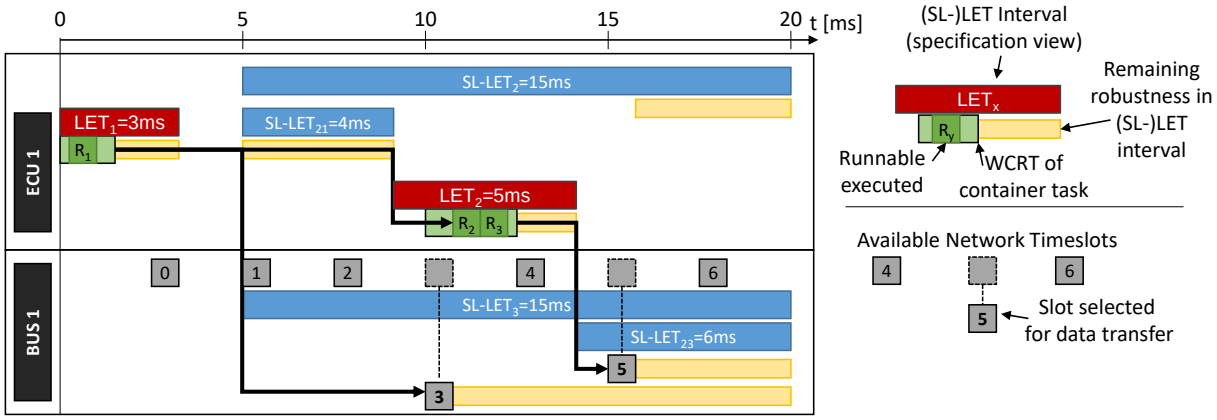

Figure 8.11: Exemplary timing of runnable execution and bus transfers with remaining robustness margins

- As an example, the container task that executes runnable R_1 has a worst-case response time of 1.5ms, which gives 50% robustness margin (relative) or 1.5ms (absolutely) compared to the 3ms interval SL-LET_1. The data flow is not affected as long as the response time does not exceed 5ms.
- The exemplary BUS0 is time-triggered and has time slots for packets each 2.5ms. The network designer in this case has to decide about a mapping of messages to time slots. For the interval SL-LET_3, slot 3 is used, which results in about 60% robustness margin. The robustness margin does not state anything about the bus or processor utilization here.
- On the other hand, the interval SL-LET_{21} is not utilized at all, since the communication between `Swc01` and `Swc02` can be realized on shared memory without any message passing. Nevertheless, SL-LET_{21} provides a slight degree of platform independence since the specification takes a possible communication latency into account a priori.
- The hierarchical decomposition of the SL-LET interval SL-LET_2 into SL-LET_{21}, SL-LET_{22} and SL-LET_{23} shows that a robustness margin in this case equals the last robustness margin in the sub-chain, namely SL-LET_{23}. The robustness margin of a composition of SL-LET intervals is not the sum of the robustness margins of the sub-intervals.

8.6.2 Modified network schedule, intra-ECU communication and WCRTs

Figure 8.12 shows a scenario where the implementation changed in multiple ways. First, `Swc01` and `Swc02` on `ECU1` are not able to use shared memory communication anymore, but the intra-ECU communication is realized by message passing. This can be for example the case when `Swc01` and `Swc02` are mapped to different processors or when the RTE does not implement shared memory communication. This consumes a non-negligible time for the processing of the RTE/ COM stack (orange box) and exploits the robustness that has been introduced a priori by the SL-LET interval SL-LET_{21}. Also

changes in the WCRT of the container task that implements runnables R_2 and R_3 can be compensated by the robustness margin of LET_2, as long as the WCRT is still smaller than LET_2. Moreover, the network schedule is changed, since the network packets are mapped to different time slots:

- The data flow is not affected in this case, since there is a time-slot available for the messages in both sub-chains that fits in the SL-LET intervals.
- This does not only comprise increased latencies such as for $\mathrm{SL\text{-}LET}_{23}$. SL-LET also preserves the data flow if the latency is decreased ($\mathrm{SL\text{-}LET}_3$). This is for example important when the network schedule is the output of an optimization algorithm that may produce completely different results for changed input requirements. Without SL-LET, an upper latency bound therefore might be fulfilled but the data flow may change if the sample is provided to early on ECU2.

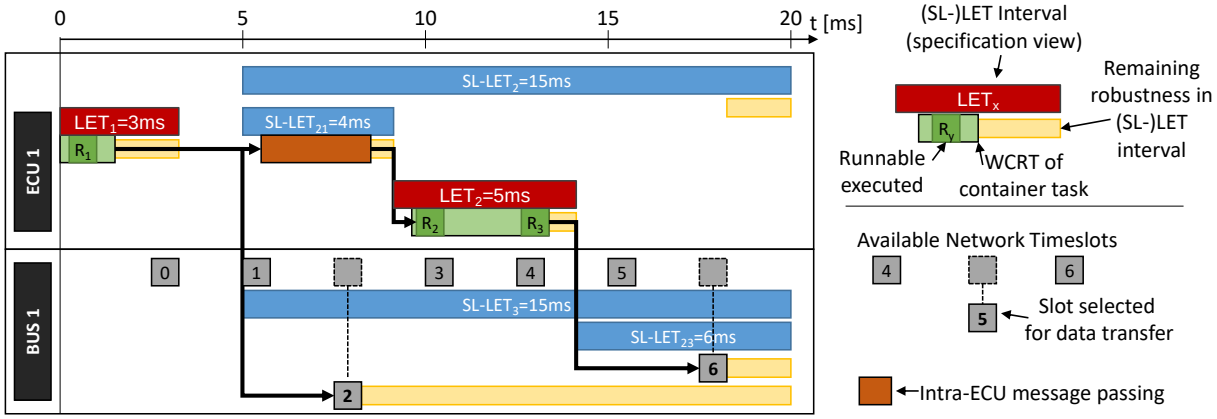

Figure 8.12: Modified timing on network due to changed network schedule with updated robustness margins

8.6.3 Updated specification

Figure 8.13 shows an example where the timing specification is modified and adapted to the non-distributed communication between `Swc01` and `Swc02`. Table 8.4 and Table 8.5 highlight the modified SL-LET specification as well as the decomposed LET intervals. This shows the strength of hierarchical compositions of SL-LET intervals, since the updated specification only affects the data flow inside of $\mathrm{SL\text{-}LET}_2$. As a result, the runnables R_2 and R_3 can be executed earlier, exploiting the shared memory communication between `Swc01` and `Swc02`. The message from `Swc02` fits in an early time slot on the bus, effectively increasing the robustness-margin of $\mathrm{SL\text{-}LET}_2$. For `Swc03`, the observable behavior remains unchanged.

SL-LET Interval	Corresponding Chain	Period	Interval Length	Offset
$\mathrm{SL\text{-}LET}_0$	*tdec01*	10ms	25ms	0ms
$\mathrm{SL\text{-}LET}_1$	*tdec02*	10ms	5ms	0ms
$\mathrm{SL\text{-}LET}_2$	*tdec03*	10ms	15ms	5ms
$\mathrm{SL\text{-}LET}_3$	*tdec04*	10ms	15ms	5ms
$\mathrm{SL\text{-}LET}_4$	*tdec05*	10ms	5ms	20ms

SL-LET_{21}	*tdec06*	10ms	(4ms) → **0ms**	5ms
SL-LET_{22}	*tdec07*	5ms	5ms	(9ms) → **5ms**
SL-LET_{23}	*tdec08*	5ms	(6ms) → **15ms**	(14ms) → **10ms**

Table 8.4: Parameters of the SL-LET intervals in Figure

LET Int.	SL-LET Int.	Runnables	Period	Interval Length	Offset to LET_1
LET_1	SL-LET_1	R_1	10ms	3ms	—
LET_2	SL-LET_{22}	R_2, R_3	5ms	5ms	(9ms) → **5ms**
LET_3	SL-LET_4	R_4, R_5	10ms	5ms	20ms

Table 8.5: Parameters of the decomposed LET intervals in Figure

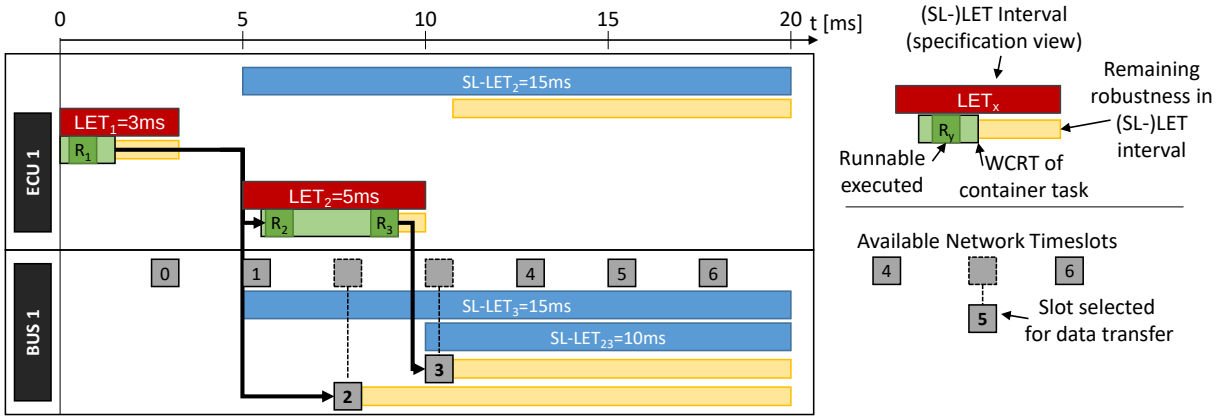

Figure 8.13: Updated SL-LET specification with new decomposition for sub-intervals

8.7 Decompose system in LET Zones with hierarchical clocks

The release and terminate event of an SL-LET interval require a synchronized time base. In contrast to the idealistic assumption, the clock of distributed ECUs might exhibit a bounded deviation, represented by a **synchronization accuracy** $\epsilon > 0$. Taking a system with two synchronized ECUs as an example, the time instant t on ECU_1 can be approximated by a time interval $[t - \epsilon, t + \epsilon]$ on ECU_2. Therefore, this use-case introduces hierarchical LET Zones to deal with this issue of imperfect synchronized clocks.

This use-case describes how *LET zones* (also called *time zones*) can be specified during the development of a distributed system. The release and termination event of an SL-LET interval provide an abstraction of physical time instances and therefore reference an abstract model clock. This clock has been assumed as a global clock in the pervious use-cases from Section 8.3 on, which means that all SL-LET intervals have been specified related to an idealistic global time base. However, synchronized time bases in a distributed system are always derivatives of a global time base and subject to a (bounded) synchronization accuracy. AUTOSAR therefore provides specification means for synchronized time bases in both, the Classic and the Adaptive Platform as

well as a specification for time synchronization protocols. This enables to implement synchronized physical clocks in a distributed system, which can act, e.g., as a basis for scheduling. LET zones on the other hand comprise abstract zone clocks, which in turn may represent a physical time base. LET zones can be specified in a hierarchical manner and provide the notion of an idealistic model time with bounded synchronization accuracy. This synchronization accuracy has to be taken into account when specifying an SL-LET interval, as it has to be covered within the SL-LET interval length. Figure 8.14 shows a straight forward example with only two hierarchical levels. Two ECUs within a vehicle are connected with Ethernet and synchronized to a global time base. Each ECU may have an individual synchronization accuracy with respect to the global time master and each ECU is represented by a LET zone in the SL-LET model. The synchronization accuracy is denoted as ϵ.

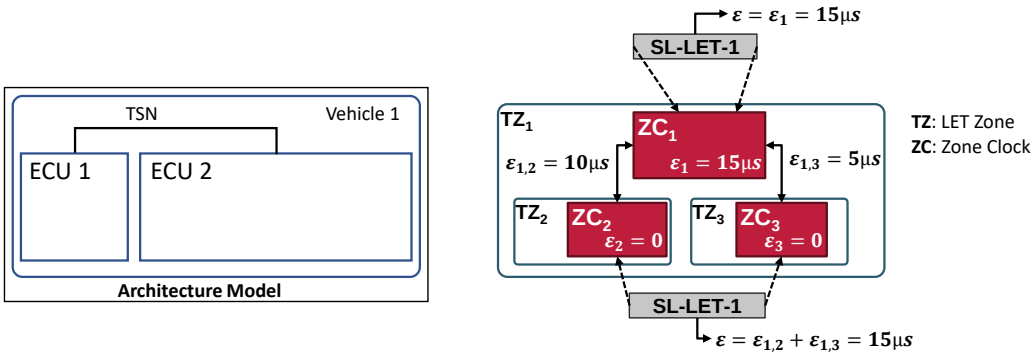

Figure 8.14: Simple notion of time zones for two ECUs within a vehicle

There are three options when specifying an SL-LET interval.

1. The synchronization accuracy can be specified between each pair of two LET zones. While this approach is obviously simple for two ECUs, it suffers from quadratic growth of the number of zone synchronization relations in a real-world system where a large number of devices resp. LET zones has to be covered. Therefore this is not explicitly denoted in Figure 8.14.

2. SL-LET therefore enables a hierarchical decomposition of LET zones, as it provides a tree based structure. The synchronization accuracy between ZC_2 and ZC_3 in Figure 8.14 can be calculated with a walk in the LET zone hierarchy and the release and terminate event can be associated with the respective LET zone. This is comparable to a white-box approach, as it takes into account the whole LET zone hierarchy.

3. A black-box approach on the other hand allows to specify SL-LET events on a higher hierarchy level. Without specifying the LET zones for both ECUs in detail, a global synchronization accuracy for the zone clock ZC_1 (which represents the vehicle-wide time base) can be specified. A lower bound for such a global accuracy is the worst case accuracy between any two sub-LET-zones.

The third approach also reflects the situation when there is no specification of hierarchical LET zones and all SL-LET events reference a global time base (resp. a global zone clock). This can be seen as a global default LET zone. Since this global zone clock

"inherits" the system-wide worst-case accuracy, it may lead to inefficiency when SL-LET events are only specified with regard to this global zone clock. Such an example is given in Figure 8.15.

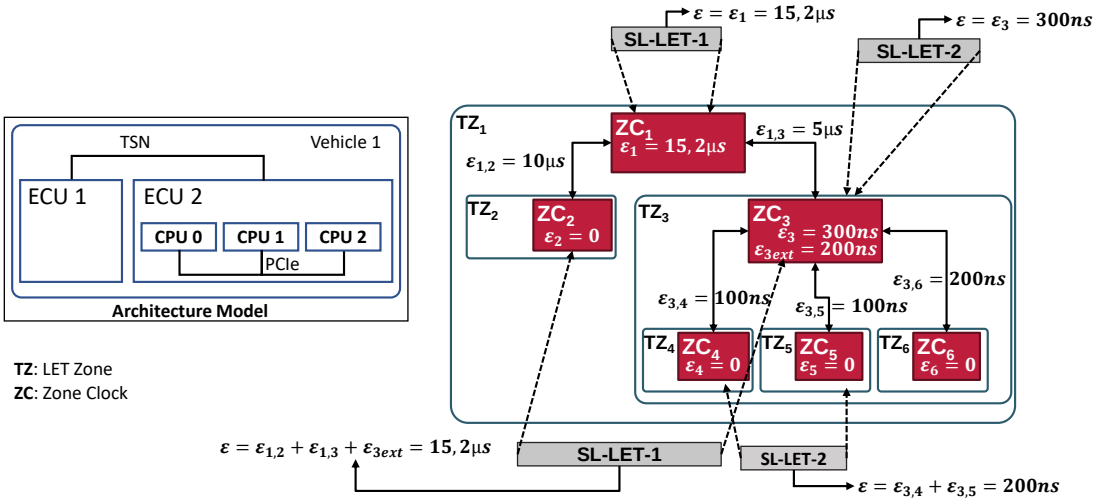

Figure 8.15: Nested LET zones with inheritance of synchronization accuracy

Within a multi-processor ECU, different processor cores are connected to an on-chip network. Each processor comprises its own time base, which is also subject to time synchronization. The synchronization within an ECU may be much better than between different ECUs and also the communication latencies over the on-chip network are lower compared to the inter-ECU network.

The interval *SL-LET-2* can be specified with respect to three different hierarchical levels. Between the zone clocks ZC_4 and ZC_5, the actual synchronization accuracy of $200ns$ provides the most accurate specification. One level above, the LET zone TZ_3 has an internal accuracy of $300ns$, which is the worst-case accuracy between any of its sub-zone clocks ZC_4, ZC_5, ZC_6. *SL-LET-2* can therefore also be specified in relation to the overall LET zone TZ_3 without much penalty. ZC_3 does not necessarily represent a physical clock within ECU 2. The model time of ZC_3 can still be used to associate SL-LET events and track the worst-case synchronization accuracy within the ECU. In contrast to that, the top-level LET zone TZ_1 has a larger synchronization error. It would also be possible to specify *SL-LET-2* with respect to ZC_1, but the large synchronization error within TZ_1 would have significant impact on the SL-LET interval length, assuming that intra-ECU communication imposes small latencies compared to inter-ECU communication.

A zone clock also has a synchronization accuracy with respect to foreign LET zones, which is the worst-case accuracy to any sub-clock. This for example has to be added for the interval *SL-LET-1*, which must take into account the accuracy between ZC_2, ZC_1 and ZC_3 as well as the worst-case accuracy to any sub-zone of TZ_3. This allows to specify an SL-LET interval between two LET zones that have an internal hierarchy, without unfolding this hierarchy. The number of hierarchy levels and the clocks that are represented by the model clocks is not limited. One can easily extend the example in

Figure 8.15 to a V2X scenario, where infrastructure devices form their own LET zones and communication ranges between an infrastructure device and an in-vehicle ECU.

8.7.1 The Role of LET Zones in the Development Process

The synchronization accuracy can be used in two ways. First, as denoted in Figure 8.15, it may act as a representation of the actual system, where the internal accuracy of a zone clock is derived from its sub-zones (bottom-up). With respect to the specification means in AUTOSAR, the opposite direction is the most important aspect, since it follows the top-down development process. For the top-down approach, a demanded synchronization accuracy in the whole system can be specified in advance and any decomposition must follow this specification.

A SL-LET specification can already be given on the functional level, but in the early development steps there is no hardware platform specified yet. Therefore, intuitively, all SL-LET events are related to a single global LET zone. Now there are different options:

A) The global LET zone for the functional model is specified assuming a perfect clock synchronization. This is an idealistic clock model with $\epsilon = 0$. Two consecutive SL-LET intervals in a cause-effect chain may be specified back-to-back, meaning that the terminate event of the producer falls together with the release event of the consumer. The data-flow within this idealistic model is unambiguous, since the data will be provided to the consumer right at the release event of the consumer. Now imagine that the producer and the consumer are mapped to different ECUs and therefore in different LET zones in the ongoing development process. With the introduction of the hardware platform, the idealistic model of $\epsilon = 0$ does not hold anymore and both, the terminate event of the producer as well as the release event of the consumer might have a bounded jitter of $\pm\epsilon > 0$. Due to the bounded synchronization accuracy, it becomes obvious that the given SL-LET specification may lead to an non-deterministic data flow. As a result, the SL-LET specification needs to be refined, ensuring sufficient clearance between the terminate event of the producer and the release event of the consumer (at least $2 \cdot \epsilon$). This clearance can be seen as a robustness against limited synchronization accuracy. A consistency check can easily detect such a violation, by comparing the SL-LET specification of the functional model (with $\epsilon = 0$) with the SL-LET specification that incorporates the hardware platform (with different LET zones and $\epsilon > 0$).

B) The global LET zone for the functional model can already be specified with an upper bound for the synchronization accuracy $\epsilon > 0$ in mind. This can act as a constraint for a later platform specification and circumvents the refinement of the SL-LET specification.

C) The functional model is not limited to comprise only one LET zone with a global synchronization accuracy. If the functional chain is already known to span over different hierarchical platform levels, this can also be reflected by multiple LET zones, which can later be refined for a specific hardware platform. An example would be V2X function that comprises sub-chains where a coarse clock synchronization can be assumed (e.g.,

on the infrastructure level) and sub-chains where a tight clock synchronization can be assumed (e.g., within the vehicle).

9 Properties and Methods for Timing Analysis

9.1 General Introduction

This section describes the general relations between timing use-cases (see chapters 4, 5, 6 and 7) and timing tasks (10). The timing properties and the timing methods are specified in details in this chapter.

The timing use-cases (for example section 6.4) presented in the former chapters (related to function level, ECU, network, or end-to-end views) usually consist of several smaller steps (listed under “main scenario” in each use case). Some of these steps are fundamentally related to timing and reappear in several use cases. We call these steps “timing related tasks” and outline them in more detail in Section 10.1.

One particularly important timing-related task is *Perform Timing Analysis*, which can be performed based on the design configuration (as in Task “Perform Model-Based Timing Analysis”) or based on an observation of the actual implementation (as in Task “Perform Implementation-Based Timing Analysis”). These timing related task can again comprise a “timing method” (see Section 9.5, which specifies in more detail how to solve this task, i.e. through simulation or static analysis.

The inputs (e.g. “the communication matrix” or “measured core execution times”) for the timing methods arise from the system specification or from observing the real system. Some of the methods deliver timing properties as an output (e.g. “worst case response time of the transmitted message”) which can be evaluated against timing constraints (for example the function may require that the frame transmission is completed in less than 10ms) during the timing task Task “Verify Timing”.

Important, but out of scope in this document is the implementation of timing methods and timing properties in tools. The approach and the timing terminology are illustrated in Figure 9.1 and 9.2.

Also important, but out of scope in this document is the concept of Logical Execution Time (LET). If an application uses the LET paradigm, each Executable Entity - which shall execute within a LET interval - has to be mapped with help of TIMEX [2] correspondingly. Further, timing analysis techniques have to be employed to ensure that all Executable Entities - which are mapped to a LET interval - terminate within the LET interval.

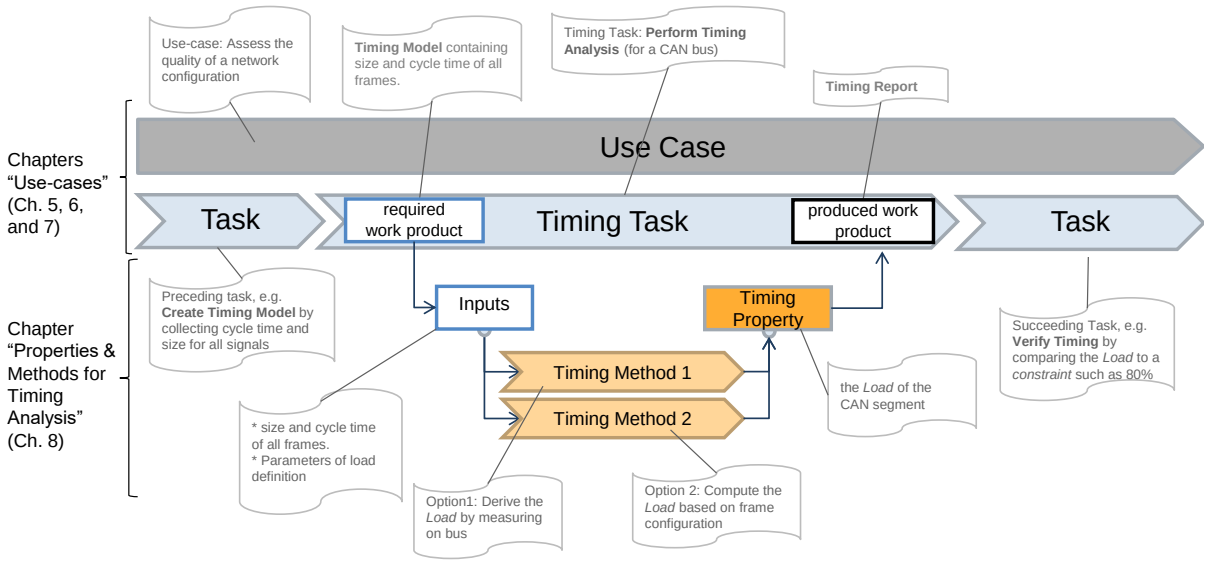

Figure 9.1: Illustration of hierarchy between use cases, timing properties, and timing methods (and related sections).

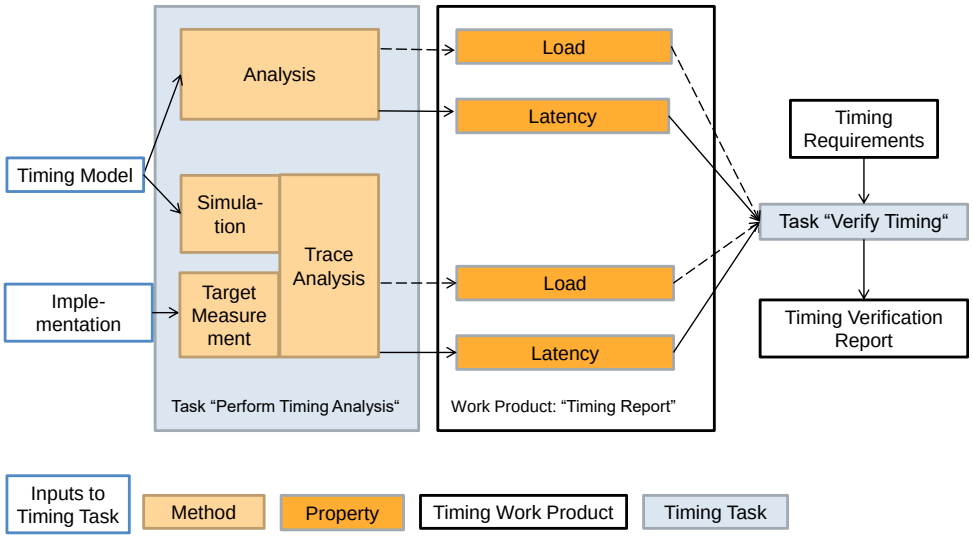

Figure 9.2: The interplay between different timing methods, timing properties and constraints

9.1.1 AUTOSAR Classic Platform Operating System

9.1.1.1 OSEK/AUTOSAR CP OS task states

AUTOSAR OS uses the scheduling concept as defined by OSEK (see "Operating System Specification 2.2.3" for details). OSEK defines task-states for two different conformance classes, BCC (Basic Conformance Class) and ECC (Extended Conformance Class). The corresponding task-state diagrams are shown in figures 9.3 and 9.4.

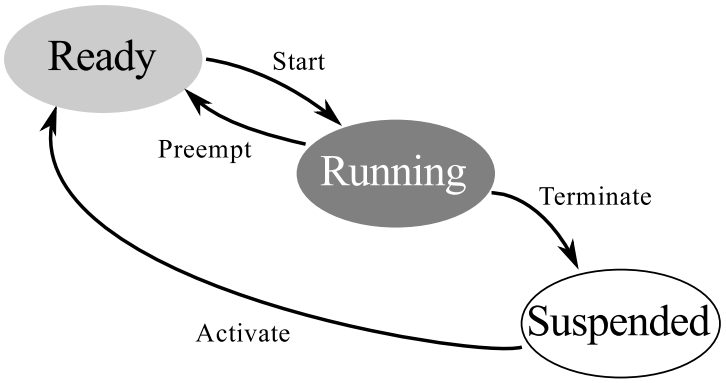

Figure 9.3: Task states and transitions as defined by AUTOSAR OS BCC

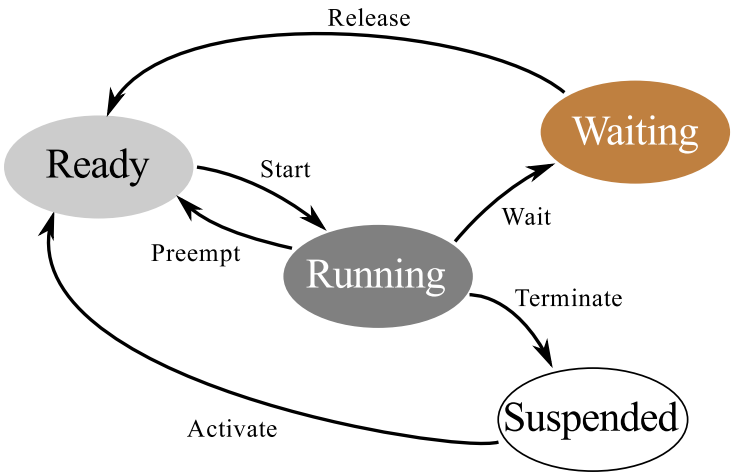

Figure 9.4: Task states and transitions as defined by AUTOSAR OS ECC

9.1.1.2 Timing parameters

Figure 9.5 shows the principle timing parameters of a task that determine its real-time behavior within a system and table 9.1.1.2 defines the symbols used. Note that the color used to indicate a task's current state at a given point in time corresponds to the color used for this state in figure 9.4.

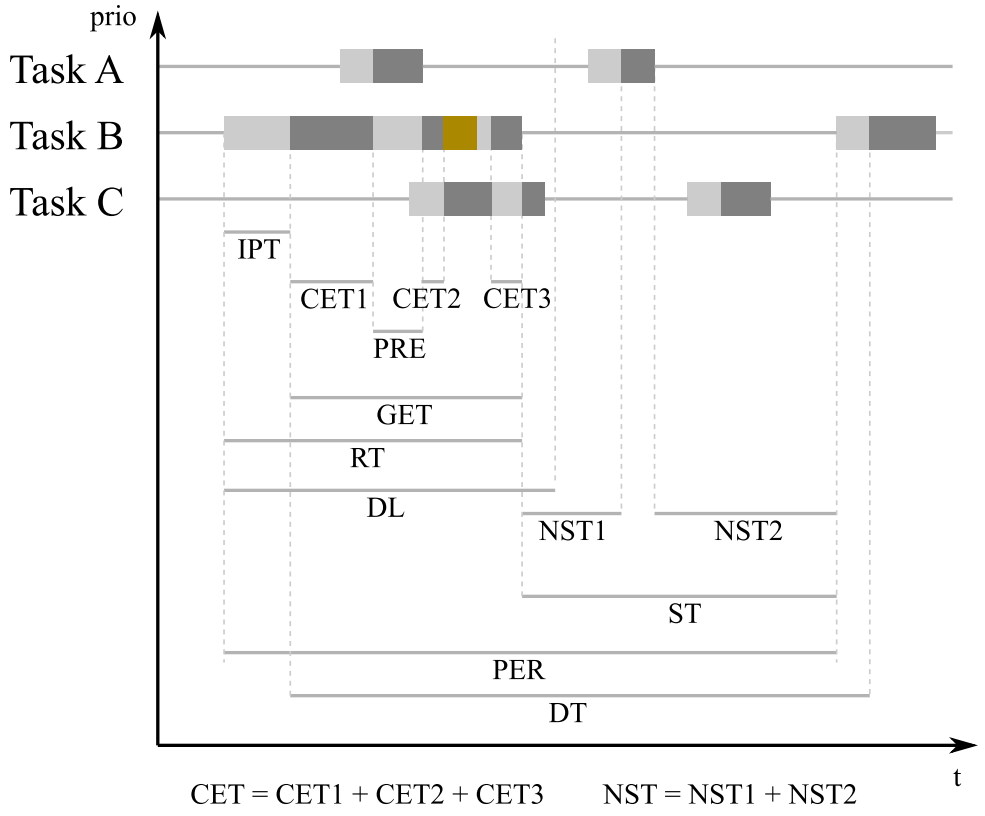

Figure 9.5: Timing parameters visualised in a trace (all related to TASK B)

ID	Abr.	Name	Description
1	IPT	initial pending time	from activation to start
2	CET	core execution time (computation time)	execution time not including any pre-emptions or “waiting” time
3	GET	gross execution time	execution time including all preemptions and “waiting” time
4	RT	response time	from activation to termination
5	DL	dead line	max. allowed response time
6	DT	delta time	from start to start (“measured period”)
7	PER	period	from activation to activation (period not as measured but as configured)
8	ST	slack time	“remaining” run-time: from termination to activation (tasks) or start (interrupts)
9	NST	net slack time	“potential additional” run-time: the ST minus all CET blocks of any TASKs or ISRs with higher priority during the ST
10	JIT	jitter	deviation of delta time from period *(not shown in the figure)*
11	PRE	Preemption time	time a task is preemted by higher priority task(s) *(not shown in the figure)*
12	CPU	CPU load	fraction of CPU time spent non-idle (usually reported in percent) *(not shown in the figure)*

Timing information

9.1.1.3 Comments on AUTOSAR OS ECC

Typically, AUTOSAR OS tasks get started and then terminate at some point in time. This is absolutely mandatory for tasks of the AUTOSAR OS basic conformance class (BCC) and should also be the case for AUTOSAR OS extended conformance class (ECC) tasks.

However, there *are* set-ups with tasks that do *not* terminate but rather loop, using WaitEvent for scheduling. This is often true for RTE tasks being generated by the RTE configuration environment. See listing 9.1 for an example. Rather than having two periodical BCC tasks – e.g. Main_Task_5ms calling CanTp_MainFunction and CanXcp_MainFunction as well as Main_Task_10ms calling CanNm_MainFunction and CanSM_MainFunction – the RTE configurator generates a non terminating ECC task and adds a second level of scheduling being controlled by WaitEvent and SetEvent.

```
1  TASK(Main_Task)
2  {
3    EventMaskType ev;
4
5    for(;;)
6    {
7      (void)WaitEvent(    Rte_Ev_Cyclic2_Main_Task_0_10ms |
8                          Rte_Ev_Cyclic2_Main_Task_0_5ms      );
9
10     (void)GetEvent(Main_Task, &ev);
11
12     (void)ClearEvent(ev & ( Rte_Ev_Cyclic2_Main_Task_0_10ms |
13                             Rte_Ev_Cyclic2_Main_Task_0_5ms      ));
14
15     if ((ev & Rte_Ev_Cyclic2_Main_Task_0_10ms) != (EventMaskType)0)
16     {
17       CanNm_MainFunction();
18       CanSM_MainFunction();
19     }
20
21     if ((ev & Rte_Ev_Cyclic2_Main_Task_0_5ms) != (EventMaskType)0)
22     {
23       CanTp_MainFunction();
24       CanXcp_MainFunction();
25     }
26   }
27 }
```

Listing 9.1: Non terminating ECC task using events for scheduling

We will not elaborate on all the disadvantages of this approach at this point but we have to address non-terminating ECC tasks and allow timing analysis also for this case. The previous definition of the CET e.g. fails. For terminating tasks (BCC as well as ECC), the CET was defined as the sum of all “running” states between the start and the termination of the task. Obviously, the CET becomes infinite if the task does not terminate.

Figure 9.6 resembles figure 9.5 but now Task B is a non-terminating ECC task. Whoever implemented the task would expect the timing properties to be computed for one “round” of the endless-loop. The gap between two subsequent rounds reflects a pseudo suspended state for Task B and thus is visualized with transparency added to the waiting state in figure 9.6.

Since the loop might include the usage of “regular” events, we now have to distinguish such “regular” events and their corresponding WaitEvent call from the events used for scheduling and *their* corresponding WaitEvent call.

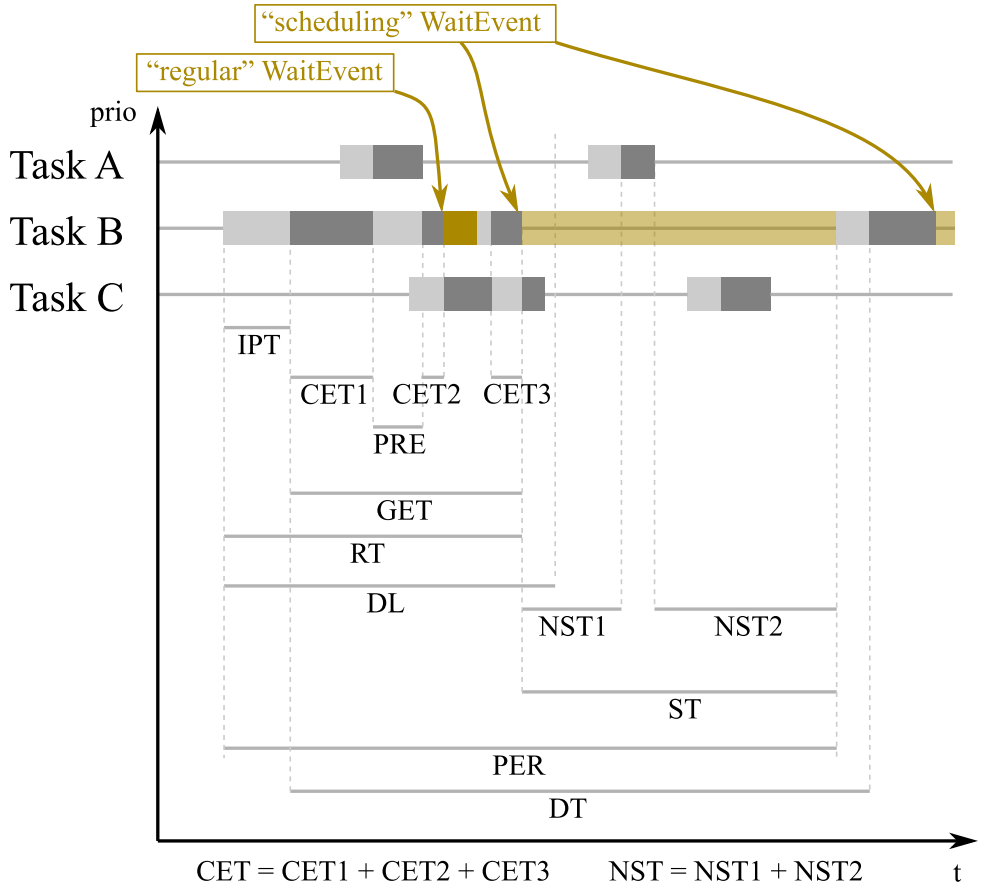

Figure 9.6: Timing parameters related to TASK B (here a non-terminating ECC task)

Listing 9.2 is derived from listing 9.1. Comments have been added for explanation and to indicate when the task changes its state. Additionally, the task now also has a “regular” event Can_Ev_TriggerSM_Main_Task. The scheduling situation shown in figure 9.6 corresponds to listing 9.2.

```
1 TASK(Main_Task)
2 {
3   // Task starts here
4   EventMaskType ev;
5
6   for(;;) // non-terminating ECC task
7   {
8     // Task "ends" here (in fact it will switch to waiting)
9     // the following WaitEvent call is a "scheduling" WaitEvent
```

```
10      (void)WaitEvent(    Rte_Ev_Cyclic2_Main_Task_0_10ms |
11                          Rte_Ev_Cyclic2_Main_Task_0_5ms      );
12      // Task "starts" here again (in fact it returned from waiting)
13
14      (void)GetEvent(Main_Task, &ev);
15
16      (void)ClearEvent(ev & ( Rte_Ev_Cyclic2_Main_Task_0_10ms |
17                              Rte_Ev_Cyclic2_Main_Task_0_5ms  |
18                              Can_Ev_TriggerSM_Main_Task      ));
19
20      if ((ev & Rte_Ev_Cyclic2_Main_Task_0_10ms) != (EventMaskType)0)
21      {
22        CanNm_MainFunction();
23      // the following WaitEvent call is a "regular" WaitEvent
24        (void)WaitEvent( Can_Ev_TriggerSM_Main_Task );
25        CanSM_MainFunction();
26      }
27
28      if ((ev & Rte_Ev_Cyclic2_Main_Task_0_5ms) != (EventMaskType)0)
29      {
30        CanTp_MainFunction();
31        CanXcp_MainFunction();
32      }
33    }
34  }
```

Listing 9.2: Non terminating ECC task using events for scheduling

The recommended task configuration for the same set-up is shown in listing 9.3. For each period – here 5ms and 10ms – it uses a dedicated task. Whenever possible, the task should be a BCC1 task. All tasks terminate.

```
1   TASK(Main_Task_10ms) // ECC
2   {
3       CanNm_MainFunction();
4       // the following WaitEvent call is a "regular" WaitEvent
5       (void)WaitEvent( Can_Ev_TriggerSM_Main_Task );
6       CanSM_MainFunction();
7       TerminateTask();
8   }
9
10  TASK(Main_Task_5ms) // BCC1
11  {
12      CanTp_MainFunction();
13      CanXcp_MainFunction();
14      TerminateTask();
15  }
```

Listing 9.3: Recommended configuration using a separate task per period

9.2 A Simple Grammar of Timing Properties

In order to avoid repeating similar definition of timing properties and methods in the following sections, this document follows a generic approach. Timing properties are

described with supporting placeholders, such as for example “<schedulable>” and “<resource>”. A “<resource>” can be e.g. a “CPU” or a “CAN bus”, and a “<schedulable>” can be e.g. the corresponding “`RunnableEntity`”, “BswSchedulableEntity” or “frame”.

Not all combinations of such terms lead to relevant/valid definitions. Therefore the actual instances are listed with the definitions. For reasons of practicality, the document however presently does not formalize the placeholder structure into a complete and consistent grammar (but such refinement may be possible in future releases).

9.2.0.1 Resources and Schedulables

<Resources> are needed to execute <schedulables>. They can schedule between several <schedulables> over time, based on an online or offline scheduling scheme. <Resources> have the capability to compute, store, transmit or receive information.

<Resources> can be divided in two categories: <unary resources>, which can execute only one <schedulable> at any given time and <multi resources> which can execute multiple <schedulables> in parallel.

A <schedulable> computes, stores, or transmits information on a <resource>. In order to make progress it must be assigned the <resource> in the scheduling process.

<Resource>	
<Unary Resource>	**Allowed <Schedulable>**
CAN bus segment	CAN frame
Single-Core CPU	Task and ISR
FlexRay Segment	FlexRay frame
Ethernet Link	Ethernet message
LIN bus	LIN frame
<Multi Resource>	
Switched Ethernet-Network	Ethernet message
Multi-Core CPU	Task and ISR

Table 9.1: Resource Overview

Note: <Multi Resources> are not covered by any of the present definitions in the document.

The timing of a schedulable is defined by its <activate> and its <terminate> events. The <activate> is the moment in time at which the <schedulable> becomes ready to perform its operation, and the <terminate> is the moment in time when it is finished.

A <schedulable> may contain <subschedulables> to differentiate between different operations.

<Schedulable>	**Allowed <Subschedulable>**
Processor Task (equivalent: ISR)	Runnable, BSW function
OS-Function	`RunnableEntity`, BswSchedulableEntity
CAN frame	PDU, Signal

FlexRay frame	PDU, Signal
Ethernet	Parameters

Table 9.2: Allowed Schedulable

9.2.0.2 Method of Derivation

The different timing properties can be derived with various methods, while not every property can be properly derived with every method (but often approximated). For example, during simulation, the message load can be observed, but it is difficult to derive the real worst-case latency. For the purpose of this document, we differentiate between the following methods (see for more details in section 9.5 :

<TimingMethod>	Explanation
Analysis	Computation or theoretical estimation of the value of the timing property
Simulation	Simulation of a system to determine the temporal development of the value of the timing property
Measurement	Measurement of a target to determine the temporal development of the value of the timing property

Table 9.3: Method of Derivation

9.2.0.3 Statistical Qualifier

Many timing properties can be tailored to different <Statistical Qualifiers>. For example, one may be interested in the average latency of a message in one case and in the maximum latency in another (for example if it is a time critical message as e.g. the total time in the active steering example). Base to do this is to determine the temporal development of the latency over the time by means of e.g. the simulation and to derive the relevant quantities like the average latency. This can be more generalized to the determination of the temporal development of an arbitrary quantity ("'x-over-Time"') and to derivation of the distribution and its momenta.

For this reason, the following <Statistical Qualifiers> are introduced:

Method	**<Statistical Qualifier>**	**<Statistical Qualifier>** derived quantity
Analysis	Best-Case	
	Worst-Case	
Simulation / Measurement	Distribution / X-over-time	Minimum
		Maximum
		Average
		Standard deviation

Table 9.4: Different Types of Timing Methods and the resulting Statistical Qualifiers

The x-over-time and the distribution depend on the related timing method, the input parameters and the boundary conditions. In contrast, the analysis approach delivers the timing property as a single value (e.g. worst-case). The (best-)worst-case denotes the state of the system with the (minimum) maximum system requirement, sometimes overestimated by the applied algorithm. However, the (minimum) maximum represents the actual observed value of the timing property here in this context.

9.2.0.4 Constraint Type

Finally, in accordance with the definition in TIMEX, the actual value of the timing property can be interpreted as a requirement (a priori to an analysis) or the worst-case can be regarded as a guarantee for the system specification (a posteriori to an analysis).

<ConstraintType>
Requirement
Guarantee

Table 9.5: ConstraintType

Figure 9.7 sketches the interplay between the value of the timing property (and its development over time and its distribution) and the constraints. The value of the timing property results from the timing method. Some of the Statistical Qualifiers are indicated on the left hand side of the distribution. Here, the guarantee results from the worst case analysis of the timing property of interest (in more general case defined in the performance specification) whereas the external requirement (defined in the requirement specification) for this timing property cannot be fulfilled in this case.

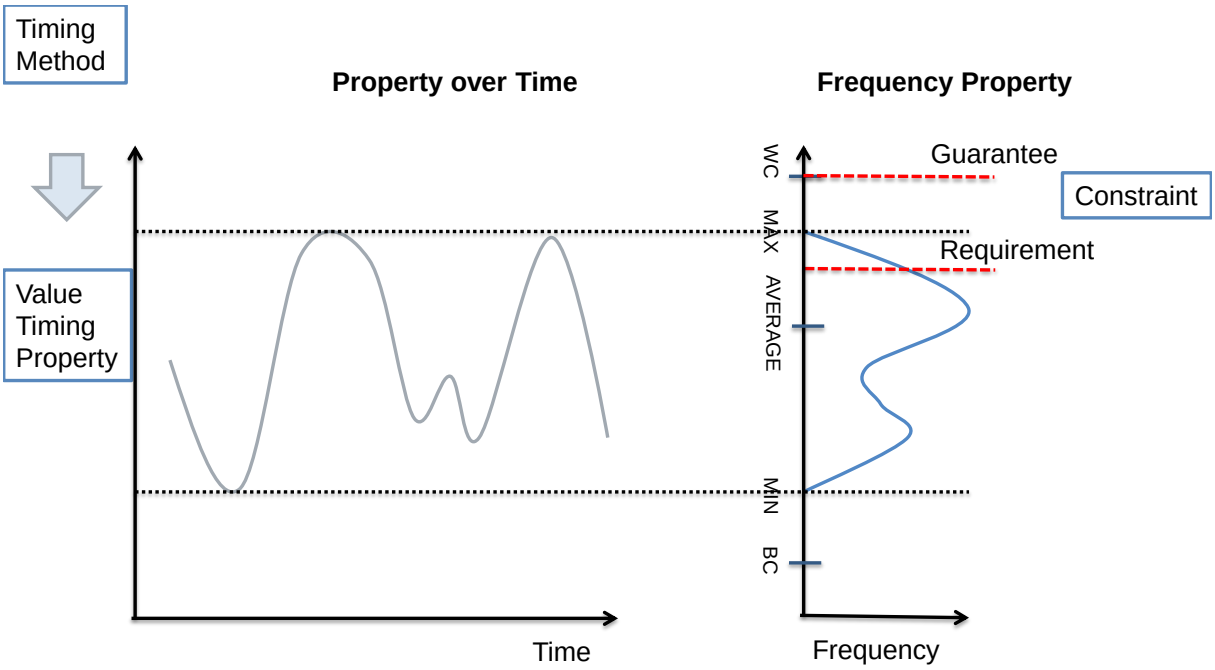

Figure 9.7: The figure illustrates the relation between the timing method, the timing property, the constraint and qualifiers (see text for more details). Here, the actual implementation does not fulfill the requirement.

9.2.1 Protocol Specifica

9.2.1.1 CAN

In order to define properties for the CAN bus the following definitions are used:

A CAN frame consists of		
Header	Standard addressing CAN Extended addressing CAN Standard addressing CAN-FD	19 bit 37 bit 22 bit
Payload	CAN CAN-FD	0..8 byte 0..64 byte
Stuff bits	Standard addressing CAN Extended addressing CAN Standard addressing CAN-FD	0..19 bit 0..25 bit 0..140 bit
Footer	CAN CAN-FD	25 bit 35 bit
Inter frame space	CAN/CAN-FD	3 bit

Table 9.6: Definition length parameter for a CAN

Summing up all parameters together yields the frame length/time (l_{frame}/t_{frame}). Thus,the general CAN properties and parameters are given by:

Scheduling	Static Priority Non-Preemptive
Activation	Periodic and/or event triggered

ID (priority)	Std. CAN/CAN-FD 0..0x7FF
Speed	CAN 100.. 1000 KBaud CAN-FD 1..10 MBaud
`Frame` length	Standard CAN 47..130 bit Standard CAN-FD 62..678 bit

Table 9.7: Definition general parameter for a CAN

For the application of the generic description to the CAN bus the following relations are applied:

Generic parameter	Actual value
<resource>	CAN bus segment
<schedulable>	CAN frame
<activate>	Event TDEventFrame.frameQueued for Transmission on sender ECU
<terminate>	Event TDEventFrame.frameTransmittedOnBus between network and receiver ECU

Table 9.8: Relation between the general and the CAN specific parameters

9.2.1.2 Activation

`Frame`	Definition
Periodic `Frame`	A frame that is activated periodically with period defined by the “cycle time”
Event-Triggered `Frame`	A frame that is activated sporadically by an external event.
Mixed `Frame`	A frame that is activated by the passing of the period or an external event. Different concepts on treating the periodic part exist (i.e. resetting of the periodic timer on arrival of sporadic events).

Table 9.9: Definitions of the frame activation

More complex activation pattern for frames in the scope of Autosar can be defined. Furthermore, OEM specific transmission modes exist.

9.3 Relations between Use Cases, Tasks, Properties and Methods

UC	Task					
	Task “Collect Timing Requirements”	Task “Create Implementation”	Task “Create Timing Model”	Task “Perform Model-Based Timing Analysis”	Task “Perform Implementation-Based Timing Analysis”	Task “Verify Timing”
Function-level use case "Identify timing requirements for a new vehicle function" (58)	x		x	x		
Function-level use case "Partition a vehicle function into a Functional Architecture" (60)	x		x			x
Function-level use case "Map a Functional Architecture to a hardware components network" (61)	x		x	x		x
ECU use case “Create Timing Model of the entire ECU” (106)	x		x			
ECU use case “Collect Timing Information of a SWE” (109)	x		x	x	x	
ECU use case “Verification of Timing” (112)				x	x	x
ECU use case “Debug Timing” (114)	x	x			x	x
ECU use case “Optimize Timing of an ECU” (117)		x			x	x
ECU use case “Optimize Scheduling” (119)		x	x	x	x	x
ECU use case “Optimize Code” (122)		x			x	x
NW use case “Integration of new communication” (83)		x	x	x	x	x
NW use case “Design and configuration of a new network” (87)	x	x	x	x	x	x
NW use case “Remapping of an existing communication link” (90)	x	x	x	x	x	x
E2E use case "Derive per-hop time budgets from End-to-End timing requirements" (66)	x					
E2E use case "Deriving timing requirements from an existing implementation" (68)	x			x	x	
E2E use case "Specify Timing Requirements for functional interfaces based on Signals/Parameters" (69)	x	x				
E2E use case "Verify guarantees against timing requirements" (72)				x	x	x
E2E use case "Trace-based timing verification of a distributed implementation" (74)					x	(x)

Table 9.10: Overview about Relation between UCs and Tasks

UC	Property			Method			
	GENERIC PROPERTY Load	GENERIC PROPERTY Latency	SPECIFIC PROPERTY Execution Time	GENERIC METHOD Determine Load	SPECIFIC METHOD Determine Load (CAN)	GENERIC METHOD Determine Latency	SPECIFIC METHOD Determine Response Time (CAN)
Function-level use case "Identify timing requirements for a new vehicle function" (58)		x	x			x	
Function-level use case "Partition a vehicle function into a Functional Architecture" (60)		x	x			x	
Function-level use case "Map a Functional Architecture to a hardware components network" (61)	x	x	x	x		x	
ECU use case “Create Timing Model of the entire ECU” (106)			x				
ECU use case “Collect Timing Information of a SWE” (109)	x		x	x			
ECU use case “Verification of Timing” (112)	x			x		x	
ECU use case “Debug Timing” (114)	x	x	x	x		x	
ECU use case “Optimize Timing of an ECU” (117)			x			x	
ECU use case “Optimize Scheduling” (119)	x		x	x		x	
ECU use case “Optimize Code” (122)			x				
NW use case “Integration of new communication” (83)	x	x		x	x	x	x
NW use case “Design and configuration of a new network” (87)	x	x		x	x	x	x
NW use case “Remapping of an existing communication link” (90)	x	x		x	x	x	x
E2E use case "Derive per-hop time budgets from End-to-End timing requirements" (66)		x	x				
E2E use case "Deriving timing requirements from an existing implementation" (68)			x	x		x	x
E2E use case "Specify Timing Requirements for functional interfaces based on Signals/Parameters" (69)	x	x	x				
E2E use case "Verify guarantees against timing requirements" (72)	x	x	x	x	x	x	x
E2E use case "Trace-based timing verification of a distributed implementation" (74)	x	x	x	x	x	x	x

Table 9.11: Overview about Relation between UCs, used Properties and applied Methods

9.4 Definition and Classification of Timing Properties

9.4.1 Classification and Relation of Properties

The properties can be grouped in two main fields: capacitive (<resource> capacity) and latency property (<schedulable> latency). Capacitive properties are the ratio of the capacity requirement by the <schedulables> to the capacity of the <resource>. Latency properties are the delays of <schedulables> due to the schedule (priority schema) on the common used <resource>.

9.4.2 Overview of regarded Timing Properties

NW/ECU	Group	Name
Generic	Capacity	GENERIC PROPERTY Load
NW	Capacity	SPECIFIC PROPERTY Load (CAN)
Generic	Latency	GENERIC PROPERTY Latency
Generic	Latency	GENERIC PROPERTY Response Time
NW	Latency	SPECIFIC PROPERTY Response Time (CAN)
Generic	Latency	GENERIC PROPERTY Transmission Time
Generic	Latency	SPECIFIC PROPERTY Transmission Time (CAN)
ECU	Latency	SPECIFIC PROPERTY Execution Time

Table 9.12: Overview about the here described Timing Properties

9.4.3 GENERIC PROPERTY Load

9.4.3.1 Scope and Application

Name	Load
Description	The load is the total share of time that a set of <schedulables> occupies a <single resource>. If the time for the occupation is calculated it can exceed the available resource time (overload). In the practical realization using simulation or measurement this scenario cannot occur. But, if the transmission load of all <schedulables> exceeding 100% (amount of send requests) is not buffered the required to transmit information can be lost or overridden.
Application	The property supports the estimation of the resource needs in ECUs and gateways and of the network, respectively.
Assumptions and Preconditions	• The time of the occupation for every individual <schedulable> is known. • The partition for the total communication amount in individual <schedulables> is done.

Relation to AUTOSAR specifications	There is no reference in TIMEX; indirect reference to AUTOSAR System Template and BSW Module Description by specifying `ExecutionTime` of an `ExecutableEntity` with the class `ResourceConsumption`.

Table 9.13: Scope, Application and Relation

9.4.3.2 Interface

Notation	$L(t, t_{window}, ...)$	
Possible<Statistical Qualifiers>	All which were mentioned in the introduction	
Parameters	t_{window}	The size of the time interval over which the load is determined. Recommended value: large but finite value
	t	The end of the time interval over which the load is determined. This parameter is required for load-over-time analysis. Default value: not specified
Range	0 to 100% (0.. infinity for calculation)	

Table 9.14: Interface

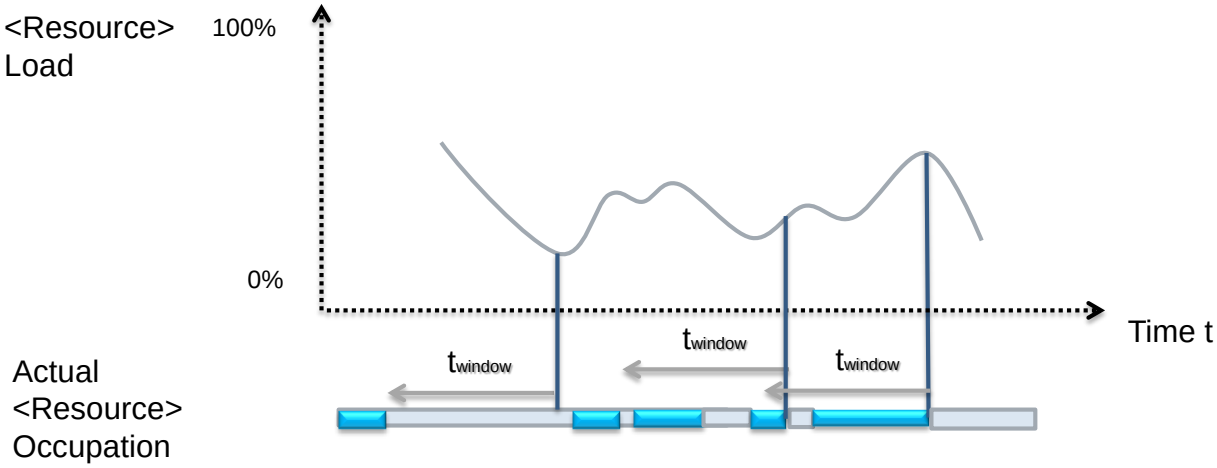

Figure 9.8: Illustration of the relation of the actual occupation and the load over time. The load $L(t, t_{window})$ is the average of the occupation over the interval t_{window} till the point in time t.

9.4.3.3 Expressiveness

The “load” indicates the overall utilization of a given <single resource>. A small load is better for stable operations due to safety and extensibility reasons. However, it shows that the <single resource> is not fully utilized, possibly missing opportunities for cost-optimization. On the other hand the remaining free resources can be used for future extension and therefor are intentionally reserved.

From perspective of real-time applications and schedulable with timing constraints, the expressiveness of load is limited. A load value below 100% allows deducing the guarantee that eventually every instance of each <schedulable> will be scheduled and executed on the <resource>. However, the completion time of a schedulable may be larger than its period or any given deadline.

Actually, the correlation to the <schedulable's> worst-case response time is small. Depending on the schedule there are examples with high load and small over-all response times and with low (but highly variable) load and high over-all response time. (compare latency, timing property worst-case response/execution time).

In [18], it was shown that given only periodic <schedulables> with deadlines equal to their periods, all <schedulable> will be serviced before their deadline if the load is smaller than 69% (independent <schedulables>). However, in practice, the presence of sporadically activated <schedulables> avoids a direct applicability of this statement.

9.4.4 SPECIFIC PROPERTY Load (CAN)

9.4.4.1 Scope, Application

Name	Bus Load (CAN)
Super Property	GENERIC PROPERTY Load
Belonging Methods	SPECIFIC METHOD Determine Load (CAN)

Table 9.15: Scope, Application, and Relation

The share of time can be calculated from the <activate> and <terminate> for the target measurement/simulation and from the frame length (see CAN) for analysis and the activation pattern (see Activation).

9.4.4.2 Interface

The Bus load (CAN) is an instance of the property GENERIC PROPERTY Load with the parameters described in Table 9.7 and 9.8.

Depending on the activation patterns, the following CAN loads are differentiated:

Periodic load	The share of time that the set of periodic frames occupies the bus.
Total load	The share of time that all frames (periodic and event-triggered, including the mixed-triggered frames) occupy the bus.

Table 9.16: Different kinds of Bus Load of a CAN Segment depending on the frame activation

9.4.4.3 Expressiveness

During run time, the CAN bus and the transmitted frames typically exhibit dynamic behavior:

- frame periods may slightly fluctuate from the specified cycle time (jitter and drift)
- the number of stuff bits depend on the actual payload
- the frame may not always carry the same amount of payload with each transmission

Depending on the selected <Statistical Qualifier> (i.e. average, maximum, ...) the properties of the CAN configuration may need to be interpreted differently due to this dynamism.

9.4.5 SPECIFIC PROPERTY Load (CPU)

9.4.5.1 Scope and Application

Name	CPU Load
Description	The general CPU load is the total share of time a CPU does not spend in an idle state over an observed time window. For a multi-core CPU, the CPU load is determined by the average load across all cores. The CPU load can also be measured on individual code parts or classes of code parts. In this case the contributions to the CPU load are calculated by the total share of time a CPU spends executing the defined code parts over an observed time window. Examples for this would be measuring the CPU resource consumption of a SW-C or AA or determining the load distribution between different tasks and servicing interrupts.
Application	The property supports the estimation of the computational resources required to execute a software on an ECU. During development it can help to track the utilization of the CPU to ensure extensibility of with new software functions or modifications of existing software functions. Estimate the required CPU resource consumption of a SWC. Justify optimization decisions to use a cheaper micro-controller or reduce the CPU frequency for lower power consumption. Detect imbalances in the software design by comparing the CPU load of system and application software or analyzing CPU load required to service interrupts.

Assumptions & Preconditions	For determining the general CPU load it is sufficient to know the idle state conditions and be able to record the entry- and exit times of the idle state. What denotes the idle state depends on the implementation. The OS may have an idle task, that is active if nothing needs to be processed, the software may be actively waiting in an endless loop for an event that requires processing or the CPU may be in a "halt" state waiting for an interrupt to trigger a processing event. For the load contribution of individual code parts, it may not be sufficient to be able to record the entry and exit times for the code parts. If a code part can be interrupt by the execution of an ISR or a task, the entry- and exit times of ISRs and activation- and deactivation times of task have to be recorded as well. If the CPU load shall be determined by class, code parts of interest need to be classified.
Relation to AUTOSAR specifications	There is no reference in TIMEX; indirect reference in CP to AUTOSAR System Template and BSW Module Description by specifying `ExecutionTime` of an `ExecutableEntity` with the class `ResourceConsumption`.

Table 9.17: Scope, Application, and Relation

9.4.5.2 Interface

Notation	$L(t, t_{window}, X, C)$	
Possible<Statistical Qualifiers>	All which were mentioned in the introduction	
Parameters	t_{window}	The size of the time interval over which the load is determined. Recommended value: large but finite value
	t	The end of the time interval over which the load is determined. This parameter is required for load-over-time analysis. Default value: not specified
	X	Definition of the idle state for the load measurement.
	C	Information on the classification of different code parts.
Range	0 to 100% (0 .. infinity for calculation)	

Table 9.18: Interface

9.4.5.3 Expressiveness

As described for GENERIC PROPERTY Load, the CPU load can only serve as a first indication for the availability of resources. Since even if the load is still relatively low it may not be possible to integrate a function with strict latency requirements.

Measurements from an implementation can include a measurement fault due to technical limitations. It may be required to insert code for the trace points or trace points cannot be added at the exact point of the event intended to be recorded. Since the CPU load is usually not required with an accuracy of more than two decimal places

and is calculated over t_{window}, where some of the faults can cancel each other out, the impact of measurement faults on the result can be neglected.

9.4.6 GENERIC PROPERTY Latency

9.4.6.1 Scope and Application

Name	Latency
Description	The latency is the amount of time between the <activate> of the first <schedulable> (it is ready to transmit on/occupy a <resource>) in a sequence of <schedulables> and the <end> of the last <schedulable> (freed from the occupation) in the list. This includes scheduling effects. If not noted otherwise, the latency refers to the processing time for one single event or one complete traversal of all <schedulables> ones. Depending on the timing property of interest and the nature of an application, two types of latency (also called "'semantics"') can be distinguished: the reaction time latency, which is the amount of time of the first reaction to a change in the input, and the data age latency, which is the amount of time that a certain input data may be processed before updated input values are available. See AUTOSAR Timing Extensions (TIMEX) [2] LatencyConstraint for details.
Application	The property supports the estimation of the resource needs and the rescheduling in ECUs, gateways and of the network, respectively. The property can be used for computation of the real-time slack of the system.
Assumptions & Preconditions	For each <resource> is known: • The access schema/arbitration strategy like bus protocol or OS scheduling • All occupation of a <resource> is error free, i.e. every utilization by the <schedulable> takes place exactly once. For each individual <schedulable> is known • The priority of the <schedulables> • The repsonse times • The triggering/activation schema including any send delay
Relation to AUTOSAR specifications	TIMEX defines the LatencyTimingConstraint of a `TimingDescriptionEventChain`.

Table 9.19: Scope, Application, and Relation

9.4.6.2 Interface

Notation	$T(t, t_{window}, X...)$	
Possible<Statistical Qualifiers>	All in the introduction mentioned	
Parameters	X	The information package for which to compute the response time
	t_{window}	The size of the time interval over which the latency is determined, i.e. the temporal interval of a trace in which the latency is determined. Default value: INF
	t	The beginning or end of the time interval over which the latency is determined. This parameter is required for X-over-time analysis. Default value: not specified
Range	0 to infinity	

Table 9.20: Interface

9.4.6.3 Expressiveness

For every hop (element) of the sequential <schedulable> list the latency per hop of the <schedulable> measures the temporal delay for its utilizations of related <single resource>. A small latency is better for stable functional operations due to safety and extensibility reasons. However, it shows that at least a part of the <resources> are not fully utilized if the latency is too small against the end-to-end deadline, possibly missing opportunities for cost-optimization. Nevertheless the latency must be smaller than the end-to-end deadline, otherwise information loss may occur. If a considerable part of <schedulables> misses their deadlines for one of the <single resource> it has not enough capacity or the schedule is not sufficiently good.

Errors during a transmission or an execution of <schedulable> may lead to a re-transmission/re-execution of specific <schedulable> which increases both the load and the latency.

The worst-case of the latency can be derived by model based formal analysis methods such as presented in [19]. By this, the latency property is conservatively computed.

The worst-case of the latency can be approximated by simulation, albeit only optimistically. The related transmission/execution requests and transmission/ execution complete events can be randomly generated and observed. The maximum of the observed values is an optimistic approximation of the worst-case latency.

When the property is derived using different methods (especially simulation/analysis and measurement) the following must be true (WC abbreviates worst case) considering only one element of the sequential <schedulable> list:

$WC\ Latency_{Analysis}$(<Schedulable>) $\geq WC\ Latency_{Simulation}$(<Schedulable>) and
$WC\ Latency_{Analysis}$(<Schedulable>) $\geq WC\ Latency_{Measurement}$(<Schedulable>)

9.4.7 GENERIC PROPERTY Response Time

9.4.7.1 Scope and Application

Name	Response Time
Description	The response time is the special case of the latency concerning only one single <schedulable>, i.e. is the amount of time between the <activate> of the <schedulable> and the <end> of the <schedulable>.This includes scheduling effects of a concurrent access to a shared <resource>. One can distinguish between a static priority pre-emptive access (in case of OSEK and other operating systems) and a static priority non-pre-emptive access (in case of CAN and most other networking systems). The response time of a <schedulable> is equal to its GENERIC PROPERTY Transmission Time or SPECIFIC PROPERTY Execution Time in the case where the resource is exclusively available to this <schedulable>. In the presence of multiple <schedulables> that are ready at the same time, the resulting response times are defined by the actual schedule.
Assumption and Precondition	For each periodic and for each mixed activation the following is known: • Period • Reference clock (optional) • Offset to reference (optional) For each event triggered and for each mixed activation the following is known: • Event model of sporadic events including minimum arrival time
Relation to AUTOSAR specifications	TIMEX defines the Response Time as LatencyTimingConstraint of a `TimingDescriptionEventChain`.

Table 9.21: Scope, Application, and Relation

9.4.7.2 Interface

The is a part of the property GENERIC PROPERTY Latency with the following parameters:

Generic parameter	**Actual value**
<resource>	Single <resource>
<schedulable>	Single <schedulable>

Table 9.22: Relation between the general latency and the response time

Notation	$T_{Response}(t, t_{window}$, <schedulable X>, <schedulables>)	
Parameter	*<schedulable X>*	The <schedulable> for which to compute the response time.
	<schedulables>	The remaining <schedulables> interacting with <schedulable X>.

Table 9.23: Interface

9.4.7.3 Expressiveness

The expression of the response time as defined is limited in some sense:

1. In the case of a large number of non-harmonic time bases, analysis time can grow beyond acceptable times. In this case, some offset relations can be ignored during analysis which may slightly decrease accuracy.

9.4.8 SPECIFIC PROPERTY Response Time (Routing)

9.4.8.1 Scope and Application

Name	Routing Response Time
Description	The routing response time is the amount of time between the reception of a message/signal from the source network until the message/signal is ready for transmission on the target network.
Application	The property supports the measurement of the routing performance of a router/gateway. This can help with the planning of time budgets for an end-to-end response time requirement or to verify the timing requirements of a router/gateway.
Assumptions & Preconditions	• The average communication scenarios are fully described. • The worst-case communication scenario is fully described. • The impact of other functions implemented on the gateway on the routing are fully known. • The routing protocol is fully described. • The buffering strategies are known. • Buffers are sufficiently large, no buffer overflows occur
Relation to AUTOSAR specifications	TIMEX defines the LatencyTimingConstraint of a `TimingDescriptionEventChain`.

Table 9.24: Scope, Application, and Relation

9.4.8.2 Interface

Notation	$T(t, t_{window}, X...)$

Possible<Statistical Qualifiers>	All in the introduction mentioned	
Parameters	X	The information package for which to compute the response time
	t_{window}	The size of the time interval over which the routing response time is determined, i.e. the temporal interval of a trace in which the latency is determined. Default value: INF
	t	The beginning or end of the time interval over which the routing response time is determined. This parameter is required for X-over-time analysis. Default value: not specified
Range	0 to infinity	

Table 9.25: Interface

9.4.8.3 Expressiveness

The routing response time for the different routing paths of a router/gateway alone, does not provide any meaningful indication on the resource utilization of the router/-gateway. A better indication for this is provided by the processing load on a router/-gateway.

The routing response time analysis can be used to verify if the router/gateway fulfills the timing requirements or helps to define time budgets for an end-to-end timing requirement, that involves a router/gateway.

If a worst-case scenario cannot be defined, model based formal analysis methods can be applied to determine an upper bound for the worst-case routing response time. Alternatively a measurement or simulation over a long time period can be used to find an optimistic approximation of the worst-case routing response time.

9.4.9 SPECIFIC PROPERTY Response Time (CAN)

9.4.9.1 Scope and Application

Name	`Frame` response time (CAN)
Description	The property provides the total time from when a frame is ready to send (<activate>, i.e. placement of the frame in an output message buffer of the CAN driver) until a frame is completely transmitted over a bus (<end>, i.e. usually leading to a Rx IRQ on a receiving ECU).
Application	The property allows assessing the communication delay of a timing critical frame. The property can be used for computation of the real-time slack (available bandwidth after accommodating all frames specified in the communication matrix).

Assumptions and Precondition	It is assumed, that • all communication on the bus is error free, i.e. every transmission takes place exactly once. • of all frames in a network that are ready to send, the CAN bus always selects the one with the lowest CAN-ID for transmission. For each frame on the bus, the following is known: • `Frame` length including stuff bits • CAN-ID • activation pattern • offsets
Relation to AUTOSAR specifications	TIMEX defines the LatencyTimingConstraint of a `TimingDescriptionEventChain`. The `TimingDescriptionEventChain` for the response time of a CAN frame can be defined as follows: • stimulus event: TDEventFrame(TDEventType=frameQueuedForTransmission) • response event: TDEventFrame(TDEventType=frameTransmittedOnBus).

Table 9.26: Scope, Application and Relation

9.4.9.2 Interface

The Response Time (CAN) is an instance of the property GENERIC PROPERTY Response Time.

Notation	$T_{Response}(t, t_{window}$, frame X)	
Parameter	*frame X*	The frame for which to compute the response time.
	stuff bits	The number of stuff bits to be assumed during analysis.

Table 9.27: Interface

9.4.9.3 Expressiveness

The expression of the response time for CAN as defined is limited in some sense:

1. Due to internal buffer structure, some CAN controllers may not be able to always provide the frame with the lowest CAN-ID (highest priority) that is ready to send to the bus arbitration. This can lead to a priority inversion with potentially larger response times than as defined by this property.

2. It is difficult to measure latency in target setups. While it is easy to identify the transmission complete events by probing the bus, the point in time when a frame becomes ready to send is more difficult (black box measurement). One option is to estimate the time by checking the bus busy time before the transmission complete event. Another option is to combine an ECU internal trace with the network trace using a reference time base.

These constraints are in part relaxed by current research such as [20], [21].

9.4.10 SPECIFIC PROPERTY Response Time (ECU)

9.4.10.1 Scope and Application

Name	Response time of a runnable entity (ECU)
Description	The property provides the total time between activating a runnable entity through termination of this runnable entity occurrence. That time span includes all times when the state is active (<running>) or passive (<ready>) until the execution is completed.
Application	The property allows assessing whether the reaction times of runnable executions are inside of an allowed time frame. This time frame usually contains a minimum and maximum border defined by a multi-dimensional time, which is the AUTOSAR element to annotate time in various units. From the response time no additionally information, such as load, can be derived.
Assumptions and Precondition	It is assumed, that • all runnable state transitions can be analyzed. • full AUTOSAR name including namespace for every runnable of interest
Relation to AUTOSAR specifications	TIMEX does provide timing description events for the element of runnable entities in the SW-C timing view. In the following a way to analyze the reaction time, which is the notation in AUTOSAR TIMEX for what is called response time in this document, is presented: • stimulus event: TDEventFrame(TDEventType=runnableEntityActivated) - referencing the runnable entity of interest • response event: TDEventFrame(TDEventType=runnableEntityTerminated) - referencing the runnable entity of interest

Table 9.28: Scope, Application and Relation

9.4.10.2 Interface

The Response Time (ECU) is an instance of the property GENERIC PROPERTY Response Time.

Notation	$T_{Response}(t, t_{window}$, runnable entity X)	
Parameter	*runnable entity X*	The runnable entity for which to compute the response time.
	event chain	The event chain used for analyzing the reaction time.

Table 9.29: Interface

9.4.10.3 Expressiveness

The expression of the response time for runnable entites in ECUs as defined can contain further information:

1. It provides information about the point in time of activating the execution context, usually the OsTask, until the runnable entity itself is executed.
2. Using the reaction time for runnable entities enables to analyze the reaction time of server runnables inside of client server interfaces.
3. To describe the reaction time of an OsTask, the above mentioned event can be used as well. By referencing the last called runnable entity of a task the reaction time of the runnable entity becomes equivalent to the OsTask reaction time it is mapped to.
4. The reaction time is highly depending on the task scheduling inside of the ECU. In order to find the reason for a certain reaction time further analysis have to be performed.

9.4.11 GENERIC PROPERTY Transmission Time

9.4.11.1 Scope and Application

Name	Transmission time
Description	The property is the special case of the response time without concerning any scheduling effects. The property provides the pure time for transmitting a <schedulable> on a single <resource> without considering any other <schedulable> on this <resource>.
Relation to AUTOSAR specifications	There is no direct constraint related to transmission times defined in TIMEX. However the transmission time could be defined similar to the ExecutionTimeConstraint by using `Frame` or PDU as referenced `ExecutableEntity`.

Table 9.30: Scope, Application, and Relation

9.4.11.2 Interface

The Transmission Time is a part of the property GENERIC PROPERTY Response Time with the following parameters:

Generic parameter	**Actual value**
<resource>	Single <resource>
<schedulable>	Single <schedulable>

Table 9.31: Relation between the general latency and the transmission time

Notation	$T_{Response}(t, t_{window}$, <schedulable X>)	
Parameter	*<schedulable X>*	The <schedulable> for which to compute the transmission time.

Table 9.32: Interface

9.4.12 SPECIFIC PROPERTY Transmission Time (CAN)

9.4.12.1 Scope and Application

Name	Transmission time (CAN)
Assumptions and Precondition	For each frame on the bus, the following is known: • `Frame` length including stuff bits
Relation to AUTOSAR specifications	There is no specific property for the transmission time in TIMEX. The definition of GENERIC PROPERTY Transmission Time can be applied directly.

Table 9.33: Scope, Application, and Relation

9.4.12.2 Interface

The Transmission time (CAN) is an instance of the property GENERIC PROPERTY Transmission Time with the parameters described in Table 9.8.

Notation	$T_{Transmission}(t, t_{window}$, frame X)	
Parameter	*frame X*	The frame for which to compute the transmission time.
	stuff bits	The number of stuff bits to be assumed during analysis.

Table 9.34: Interface

9.4.13 SPECIFIC PROPERTY Execution Time

9.4.13.1 Scope and Application

Name	Execution Time (ET)
Description	The execution time indicates a time required for a certain computation. In this context a computation can be a runnable, a subfunction or just a sequence of commands.
Goal	This property is a required input information for run time budgeting and the ECUs schedule feasibility.
Assumptions	n/a

Relation to TIMEX	TIMEX defines an ExecutionTimeConstraint of an `ExecutableEntity`. The SPECIFIC PROPERTY Execution Time as defined above corresponds to the executionTimeType "'gross"', i.e. calls to external functions are included.

Table 9.35: Scope, Application, and Relation

9.4.13.2 Interface

The Execution Time is a part of the property GENERIC PROPERTY Response Time.

Output	The scalar result value is usually stated in micro-, milli- or nanoseconds.
Range	0 to infinity

Table 9.36: Interface

9.4.13.3 Expressiveness

For hard real-time systems an important statistical qualifier (also see subsubsection 9.2.0.4) is the worst case execution time (WCET) that is required to complete a certain computation. The WCET is an indicator for resource consumption usually a predefined value must be reached or derived. To predict and proof the correct software execution the WCET is an important property. In practice it is recommended to use different timing methods to determine the WCET in order to gain the confidence of the result. These methods are static, dynamic and hybrid approaches. If it is not possible to determine the WCET in the field an upper safe limit needs to be used as equivalent. Based on the worst case execution time of several computations one or more WCRT (worst-case response time) might be determined which in most cases are more relevant.

For ECU use case "Optimize Code" next to the WCET the average execution time can be interesting. Huge differences between the both of them or the average execution time and the maximum execution time usually indicate optimization potential.

9.5 Definition, Description and Classification of Timing Methods

9.5.1 Classification and Relation of Methods

Roughly, the methods can be grouped in three main fields: simulation, analytical calculation and measurement. Another criterion to distinguish methods is to consider the origin of the data: model-based or measurement-based. This classification is closely related to the moment in which stage of the timing process the method can carry out (in the specification phase or verification phase).

9.5.1.1 Analytical calculation

Static Code Analysis works on the source code or binary code level of an executable software or part of it. A distribution of SPECIFIC PROPERTY Execution Time is determined. Therefore the call graph and the instruction sequence is reconstructed and analyzed. A lower limit for the BCET (best case execution time) and the upper bound for the WCET (worst case execution time) is calculated for a given code fragment (e.g. a function) by applying Statistical Qualifier. Beside the software that should be analyzed, symbol information and annotations for additional constraints (e.g. build options, range of input values, integration/hardware specific constraints) must be provided. Any real core execution time is guaranteed to be within this interval, as long as this fragment is not interrupted. Furthermore, any data present only at run-time (e.g. upper bounds on the loop iterations and the content of dynamic function pointers) has to be provided manually in the form of additional annotations. For a proper static code analysis the target hardware behavior must be known in detail (e.g. access time for different memory areas, caching, and so forth). In modern systems the behavior model can be quite complex and therefore limitations regarding the results precision may occur. The results of static code analysis should be validated with the results from alternative methods described in section 9.5.

Scheduling analysis Based on the model of a certain scheduler (e.g. a certain RTOS), scheduling analysis tools take minimum/maximum core execution times and an application model as input and provide e.g. the guaranteed WCRT. This allows checking whether any deadlines will be missed under the given conditions. For each task's and interrupt's worst case, a trace is generated allowing to analyze the run time situation under which it occurs. The execution times fed into the analysis can be either budgets, estimations, or outputs from other tools, e.g. statically analyzed BCET/WCET or traced/measured data. Thus, scheduling analysis allows to verify new concepts without implementing them as an advandage. Furthermore existing concepts can be amended for concept verification or solution space exploration.

Network analysis Network analysis for a single network segment computes the worst-case response time for each frame/package transferred via the network. This is usually possible based on the same type of design data that is needed to configure the connected ECUs (e.g. AUTOSAR System Extracts). The main information is the (gross) size of each frame/package (e.g. based on the size of the contained signals/parameters and the protocol header), the frame's transfer properties (i.e. its cycle time or debounce time of external triggers), and of course the transmission speed of the network. The analysis takes conflicts on the networks and synchronization between frames into account when computing the worst-case response times. This basic result can be aggregated into a complete timing profile for a bus or gated network. In case of highly dynamic timing behavior network analysis can be mixed with measurement-based approaches by replacing model-based design data with event traces from actual measurements.

Compositional analysis Compositional analysis allows to consider an activity chain consisting of different <schedulable> on different <resources>. It adds the chained response time of different <schedulable> on one <resource> in a first step and then

the response time of the resources. If a worst case consideration is made, this method can be very conservative as in reality the probability of a worst case response time on several chained resources is by far lower than the probability of a single resource worst case, which by itself is conservative.

9.5.1.2 Simulation

In general, a simulation needs enough runs (simulation time) to ensure a statistical relevance of the results and to cover the parameter space of all degrees of freedom (e.g. the jitter of the send requests, the sending arrangement of the frames).

Code simulation Code simulators simulate the execution of given binary code for a certain processor. A wide variety of code simulators exist. Simple instruction set simulators provide very limited information about the execution time whereas complex simulators consider also pipeline- and cache-effects. To achieve reliable WCET information from a code simulator, it has to be embedded into a test environment which actually causes the worst case to be simulated.

Scheduling simulation Scheduling simulators provide similar functionality as the scheduling analysis. Instead of calculating the results, they simulate run time behavior. The observed timing information and generated traces are the main output. If the worst case scenarios are simulated, the observed response times will equal the WCRTs. Some simulators allow task definitions in C language so that complex applications models are supported while offering a specification language well known to automotive engineers.

Network simulation Network simulation is the technique of predicting the actual timing of a bus segment or network of segments based on models of the actual configurations. These models are typically derived from the same design data is needed to configure the connected ECUs (e.g. AUTOSAR System Extracts). The main information is the size of each frame/package (e.g. based on the size of the contained signals/parameters and the protocol header), the frame's transfer properties (i.e. its cycle time or debounce time of external triggers), and of course the transmission speed of the network. The network simulator is specific to a particular network protocol and will typically create random traffic within the bounds specified by the model data and unroll specific schedules. These schedules can be investigated with respect to resulting frame response times, network load and so on. As another kind of network simulation the remaining bus simulation is not a timing specific method, but many timing issues like arbitration latency, jitter, high load behaviour, etc. can be carried out on a real physical layer for experimental purposes. As a result other simulation methods can be verified.

Processor-In-The-Loop Simulation (PIL) is used to determine timing properties like SPECIFIC PROPERTY Execution Time or GENERIC PROPERTY Load of a specific software system. Therefore the compiled software will be executed in the embedded target processor on an evaluation board, a prototype hardware or the actual ECU. In order to be able to execute the software correctly the required run-time environment

will be simulated. The simulation platform stimulates and calls the software under investigation. During the execution the required output data is captured. The output data is analyzed to derive the required timing properties. To carry out a PIL the analyzable executable (e.g. elf file) and input vectors for stimulation must be provided. The results of the PIL simulation should be validated with the results from alternative methods described in section 9.5.

The input stimuli vector which will be used for the PIL needs to stimulate the software in a way that the highest physically possible code coverage is reached. The quality of the input stimuli vector shall be shown in a separate “input stimuli vector acceptance test” which proofs an appropriate coverage. The accuracy of the result strongly depends on the quality of the input stimuli vector.

The tracing solution which captures the output data must have the capability to measure the execution time between defined profiling points. Profiling points define the start and end point for the measurement.

Referencing Use-cases	• ECU use case “Collect Timing Information of a SWE” • ECU use case “Verification of Timing” • ECU use case “Optimize Timing of an ECU” • ECU use case “Optimize Scheduling” • ECU use case “Optimize Code”
Referencing Timing Properties	• SPECIFIC PROPERTY Execution Time • GENERIC PROPERTY Load • GENERIC PROPERTY Response Time

Table 9.37: Relation

Discrete-Event-Simulation (DES) is used to simulate the dynamic behavior of the system. It models the operation of a system as a discrete sequence of events in time. Each event occurs at a particular instant in time and marks a change of state in the system. The method can be applied whenever a timing model of the system is available. The results of this method are timing properties of the system. A Table 10.12 of the system must be available and the accuracy of the result strongly depends on the quality of the input model.

Referencing Use-case	• ECU use case “Collect Timing Information of a SWE” • ECU use case “Create Timing Model of the entire ECU” • ECU use case “Verification of Timing” • ECU use case “Optimize Scheduling”
Referencing Timing Properties	• GENERIC PROPERTY Response Time • GENERIC PROPERTY Load • SPECIFIC PROPERTY Execution Time

Table 9.38: Relation

Hardware-In-The-Loop Simulation (HIL) can be used to determine timing properties like GENERIC PROPERTY Response Time or GENERIC PROPERTY Load of a specific ECU software.

To carry out a HIL simulation the software must be integrated to the actual ECU. The ECU is connected to a so called Hardware-In-The-Loop simulator which is able to simulate car’s environment that is required for the proper functionality of the ECU. During the simulation the desired output data is captured. The output data is analyzed to derive the required timing properties.

The input stimuli vector needs to stimulate the ECU software in a way that the highest physically possible code coverage is reached. The accuracy of the result strongly depends on the quality of the input stimuli vector.

The tracing solution must have the capability to measure the execution time between defined profiling points. Profiling points define the start and end point for the measurement.

Referencing Use-case	• ECU use case “Collect Timing Information of a SWE” • ECU use case “Verification of Timing”
Referencing Timing Properties	• GENERIC PROPERTY Response Time • GENERIC PROPERTY Load • SPECIFIC PROPERTY Execution Time

Table 9.39: Relation

9.5.1.3 Measurement and Tracing

Measurement on ECU level Timing measurement is often based on hook routines which are invoked by the RTOS. The real system is analyzed and the observed timing information is provided.

Measurement on Network level The timing measurement is done by special hardware connected to the hardware of the real network. Depending on the protocol and the applied measurement device the time stamp is imprinted at different point in time during the transmission of the relevant <schedulable>. The accuracy is given by the tracing device and shall fulfill the sampling theorem.

Tracing observes the real system. Tracing means persistent recording of measurement data streams. This can be recording of discrete events or sampled and quantized data from time contiguous sources in combination with a time stamp. For dedicated events, time stamps together with event information is placed in a trace buffer. The selection of events can be very fine grained like for flow traces which allow reconstructing the execution of each machine instruction or coarse grained like when tracing scheduling related events only. Tracing can base on instrumentation (i.e. software modification) or on special tracing hardware. Traces can be visualized and analyzed offline, e.g. for debugging purposes. Different kinds of timing information can be extracted from a trace. Sometimes an implicit protocol overhead has to be included for the correct computation (e.g. stuff bits for load computation on CAN).

9.5.1.4 Determination of the Comparability of the Different Methods

Comparing analysis on one hand and simulation/measurement on the other hand the loads shall be coincident in the long-time limit (under identical boundary conditions). The difference vanishes if all parameters are chosen in the same manner. In general, the simulation and the observation yield an optimistic approximation in the same manner depending on the sample/probe size (measurement/simulation time).

In order to compare the results of different methods (especially simulation/analysis and measurement) a check that all <schedulables> are contained in the output is highly recommended.

9.5.2 Overview of regarded Methods

The fields for all methods are analysis, simulation and measurement.

NW/ECU	Group	Name
Generic	Load	GENERIC METHOD Determine Load
NW	Load	SPECIFIC METHOD Determine Load (CAN)
Generic	Latency	GENERIC METHOD Determine Latency
NW	Latency	SPECIFIC METHOD Determine Response Time (CAN)

Table 9.40: Overview of regarded Methods

9.5.3 GENERIC METHOD Determine Load

9.5.3.1 Scope and Application

Description	The method yields the load (distribution) over a defined time interval.
Reasoning	The method supports the estimation of the resource needs in ECUs, gateways and of the network, respectively.

Table 9.41: Scope and Application

9.5.3.2 Classification

System	NW / ECU
Applied Protocol	CAN / FlexRay / OSEK / AUTOSAR etc.
Approach	Analysis / Simulation / Measurement

Table 9.42: Classification

9.5.3.3 Relation

Requirements	Interface input and boundary conditions (see Table 9.44)
Process Steps	The method shall be applied during the following process steps: • Verification of a software implementation / of data definition and of the configuration of communication networks • Requirement analysis for further development • Resource optimization during development phase

(Pre) Timing Property	Depending on implementation this method requires the timing properties transmission time and/or execution time of all <Schedulables> on the considered <Resource> (e.g. GENERIC PROPERTY Transmission Time and SPECIFIC PROPERTY Execution Time)
Belonging Post Timing Property	GENERIC PROPERTY Load

Table 9.43: Relation

9.5.3.4 Interface

Input	The method requires parameters such as: • <Schedulables> (e.g. tasks/frame/PDUs) with their overall times (transmission time, execution time), their activation pattern (e.g. periodic/cyclic, sporadic) and potentially other parameters (e.g. stuff-bits in case of CAN Bus communication) • Transmission/execution speed of the regarded <single resource> (e.g. CAN bus speed or processor speed) • Model of the spontaneous occurrence of <schedulable> (e.g.event-triggered frames) / approximation of the occurrence of the spontaneous events
Boundary condition, Settings and Variants, Precondition	• Environmental states (like driving states)
Output	The result of this method is the load on a <resource> (NW/ECU) captured by the timing property GENERIC PROPERTY Load.

Table 9.44: Interface

9.5.3.5 Implementation

The implementation of the method for deriving the load of a network or of an ECU depends on the considered approach, namely analysis, simulation or measurement. Implementation details can be found in the corresponding specific methods.

9.5.4 SPECIFIC METHOD Determine Load (CAN)

9.5.4.1 Scope and Application

Brief Description	The method yields the load (distribution) on a CAN bus over a defined time interval.
Reasoning	The method supports the determination of the resource needs of a CAN network.

Table 9.45: Scope and Application

9.5.4.2 Classification

System	NW
Applied Protocol	CAN
Approach	Analysis / Simulation / Measurement

Table 9.46: Classification

9.5.4.3 Relation

Requirements	Interface input (see Table 9.48)
Process Steps	The method shall be applied during the following process steps: • Verification of data definition and of the configuration of a CAN bus • Requirement analysis for further development • Resource optimization during development phase
(Pre) Timing Property	The method requires the SPECIFIC PROPERTY Transmission Time (CAN) of all frames on the considered CAN bus.
Belonging Post Timing Property	SPECIFIC PROPERTY Load (CAN)

Table 9.47: Relation

9.5.4.4 Interface

The specific method Determine Load (CAN) is an instance of the GENERIC METHOD Determine Load.

Input	The method requires the CAN parameters defined in 9.7 and 9.8.
Output	The result of this method is the load on a CAN bus captured by the SPECIFIC PROPERTY Load (CAN).

Table 9.48: Interface

9.5.4.5 Limitation in Application

At the moment, there is no established treatment for the spontaneous occurrence of event-triggered frames. Therefore, a unified model or activation pattern for the spontaneous occurrence has to be applied in order to achieve comparable results between different configurations.

A general treatment for the calculation of stuff-bits is missed. Therefore, different assumptions regarding the number of stuff bits shall be considered, i.e. a minimal, an average and a maximal number of stuff-bits.

9.5.4.6 Implementation

9.5.4.6.1 Analysis

The method for deriving the bus load for a CAN bus by analysis is based on mathematical formulas. These formulas can be implemented in a tool which supports the import of input parameters, the calculation of the load values and the export of the results.

The method has to enable the calculation of optimistic (best-case), average and pessimistic (worst-case) bus load values. For that purpose, different assumptions regarding the number of stuff-bits shall be implemented, i.e. a minimum number (for the optimistic approach), an average number, and a maximum number (for the pessimistic approach). Furthermore, different models of the event-triggered frame activation patterns shall be supported. The derivation of the load by analysis takes into the consideration the cyclic events with their periods and the spontaneous events with an event model. For example, the spontaneous events can be modeled with their debounce times as a "cycle" or with their maximum occurrence rate. Depending of the pessimistic or optimistic approach the calculation can estimate the upper bound with the lower limit of the period or with a specified period for the latter one.

The formula to calculate the bus load includes the pessimistic/optimistic approach depending on estimation of the stuff-bits for the analysis (see the formula for the stuff bits below, for CAN frames with 29-Bit Identifier there are deviations). The CAN parameters are given in 9.6.

$$t_{frame} = t_{\text{stuff bits(frame)}} + (47 + 8 * payloadlength(frame)[Byte]) * \tau_{Bit} \quad (9.1)$$

$$L(t_{frame}, t_{cycle(frame)}, payloadlength(frame)) = \sum_{frame} \frac{t_{frame}}{t_{cycle(frame)}} \quad (9.2)$$

Whereas payload length (in Byte) is the length of the data part of the CAN frame, t_{bit} is the time for the transmission of one bit, t_{cycle} is the specified period.

Alternatively, it follows for CAN-FD (for frame length larger than 16 Byte)

$$t_{frame} = t_{\text{stuff bits(frame)}} + 30 * \tau_{Bit} + (30 + 8 * payloadlength(frame)[Byte]) * \tau_{Bit}^{high} \quad (9.3)$$

where τ_{Bit}^{high} is the high data rate of the bus.

This approach estimates the bus load generated by the periodic messages on a bus during an infinitely long time window (t_{window} is infinity, the present point in time t does not play any role). The time for a frame is maximized due to a conclusion of all possible stuff bits. The event-triggered frames are neglected.

For CAN, different assumptions for stuff bits shall be implemented (minimal, average, maximal). Depending on the implemented approach, the calculation shall include a

minimum (optimistic approach), an average or a maximum (pessimistic approach) number of stuff-bits. For each frame the following calculation formula for the maximal stuff-bit time shall be used. The average number of stuff-bit time can be derived by dividing by 2.

$$t_{\text{stuff bits(frame)}} = \left\lfloor \frac{34 + 8 * payloadlength(frame)[Byte]}{4} \right\rfloor * \tau_{Bit} \quad (9.4)$$

The equivalent formula for CAN-FD (for frame length larger than 16 Byte) is

$$t_{\text{stuff bits(frame)}} = 4 * \tau_{Bit} + \left[7 + \left\lfloor \frac{5 + 8 * payloadlength(frame)[Byte]}{4} \right\rfloor \right] * \tau_{Bit}^{high} \quad (9.5)$$

9.5.4.6.2 Simulation

Every frame is simulated with its individual activation pattern (periodic, event triggered or mixed activation). Furthermore even for event triggered frames, different models for their activation patterns shall be supported. Different payloads may lead to different numbers of stuff bits which have to be considered for the computation of the frame time. In the simulation all frames try to access the network at their trigger points in time, but only the frame with the highest priority (lowest ID) gains the access to the bus. Regarding a temporal averaging interval t_{window} the bus load is given as a ratio of the time for the sending of all frames to this interval:

$$L(t_{frame}, t_{window}, t) = \sum_{frame \in t_{window}} \frac{t_{frame}}{t_{window}} \quad (9.6)$$

where t_{frame} is the time for each individual frame.

9.5.4.6.3 Measurement

The formula to calculate the bus load for CAN from a measurement is equal to the formula of the simulation and given by:

$$L(t_{frame}, t_{window}, t) = \sum_{frame \in t_{window}} \frac{t_{frame}}{t_{window}} \quad (9.7)$$

t_{frame} is again the time for each individual frame. The result is strongly dependent on the averaging (measurement) interval t_{Window}. In the short time the limes of the load can reach 100%. Important is to include the stuff bit overhead for the correct computation of the frame time and therefor of the load.

9.5.5 GENERIC METHOD Determine Latency

9.5.5.1 Scope and Application

Brief Description	The method yields the latency of <schedulables> when executed on <resources>.
Reasoning	The method supports the estimation of the resource needs in ECUs and gateways and of the network, respectively.

Table 9.49: Scope and Application

9.5.5.2 Classification

System	ECU / Network
Applied Protocol	CAN / FlexRay / OSEK / AUTOSAR etc.
Approach	Analysis / Simulation / Measurement

Table 9.50: Classification

9.5.5.3 Relation

Requirements	Interface input, see Table 9.52.
Process Steps	The method shall be applied during the following process steps: • Verification of an software implementation / of data definition and of the configuration of communication networks • Requirement analysis for further development • Resource optimization during development phase
(Pre) Property	Depending on implementation this method requires the timing properties transmission time and/or execution time of all <Schedulables> on the considered <Resource> (e.g. GENERIC PROPERTY Transmission Time and SPECIFIC PROPERTY Execution Time).
Belonging Post Property	GENERIC PROPERTY Latency

Table 9.51: Relation

9.5.5.4 Interface

Input	The method requires parameters such as: • Implementation of the software, analyzable executable (e.g. elf file), input vectors for stimulation a.s.o. • <Schedulables> (e.g. tasks/frame/PDUs) with their overall times (transmission time, execution time), their activation pattern (e.g. periodic/cyclic, sporadic) and other parameters (e.g. stuff-bits in case of CAN communication) • Scheduling/priority rules (e.g. preemptive, non-preemptive, mixed-preemptive) on the <resource> • Transmission/execution speed of the regarded <resource> (e.g. CAN bus, processor speed) • Model of the spontaneous occurrence of <schedulables> (e.g. event triggered frames) / Approximation of the occurrence of the spontaneous events
Boundary condition, Settings and Variants, Precondition	• Environmental states (like driving states)
Output	The result of this method are timing properties of type GENERIC PROPERTY Latency for all <schedulables> (frames/tasks) on a <resource> (NW/ECU).

Table 9.52: Interface

9.5.5.5 Implementation

The implementation of the method for deriving the latencies of <schedulables> on <resource> (i.e. networks or ECUs) depends on the considered approach, namely analysis, simulation or measurement. Implementation details can be found in the corresponding specific methods.

9.5.6 SPECIFIC METHOD Determine Response Time (Routing)

9.5.6.1 Scope and Application

Brief Description	The method yields the response time for routing a message from one network to another.
Reasoning	The response time for an event chain across networks can be significantly impacted by the delay caused by routing a message from one network to another.The method supports the determination of the delay caused by routing messages or to measure the routing performance of a router/gateway.

Table 9.53: Scope and Application

9.5.6.2 Classification

System	Network
Applied Protocol	Unspecific
Approach	Analysis / Simulation / Measurement

Table 9.54: Classification

9.5.6.3 Relation

Requirements	Interface input, see Table 9.56.
Process Steps	The method shall be applied during the following process steps: • Verification of an software implementation / of data definition and of the configuration of communication networks • Requirement analysis for further development • Resource optimization during development phase
(Pre) Property	Depending on implementation this method requires the timing properties transmission time and/or execution time of all <Schedulables> on the considered <Resource> (e.g. GENERIC PROPERTY Transmission Time and SPECIFIC PROPERTY Execution Time).
Belonging Post Property	SPECIFIC PROPERTY Response Time (Routing)

Table 9.55: Relation

9.5.6.4 Interface

Input	The method requires parameters such as: • Implementation of the router/gateway software, analyzable executable (e.g. elf file), input vectors for stimulation a.s.o. • <Schedulables> (e.g. tasks/frame/PDUs) with their overall times (transmission time, execution time), their activation pattern (e.g. periodic/cyclic, sporadic) and other parameters (e.g. stuff-bits in case of CAN communication) • Scheduling/priority rules (e.g. preemptive, non-preemptive, mixed-preemptive) on the <resource> • Transmission/execution speed of the regarded <resource> (e.g. CAN bus, processor speed) • Model of the spontaneous occurrence of <schedulables> (e.g. event triggered frames) / Approximation of the occurrence of the spontaneous events • Routing protocol/buffering strategy description
Boundary condition, Settings and Variants, Precondition	• Environmental states (like driving states)

Output	The result of this method are timing properties of type SPECIFIC PROPERTY Response Time (Routing) for one or more routing path(s).

Table 9.56: Interface

9.5.6.5 Limitation in Application

At the moment, there is no established treatment for the spontaneous occurrence of event-triggered frames. Therefore, a unified model or activation pattern for the spontaneous occurrence has to be applied in order to achieve comparable results between different configurations and between the applied approaches i.e. analysis/simulation/measurement.

9.5.6.6 Implementation

The implementation of the method for deriving the routing latencies depends on the considered approach, namely analysis, simulation or measurement.

9.5.6.6.1 Analysis

In case of using an analytical approach the routing latency derivation is based on mathematical formulas. These formulas can be implemented in a tool which supports the import of input parameters, the calculation of the routing response time values and the export of the results.

9.5.6.6.2 Simulation

From the network side only frames from the source networks need to be simulated, that are processed by the router/gateway. This include messages that have to be actively filtered by the router/gateway and messages processed by a function on the gateway, if it takes away from the processing time for routing messages. From the gateway side message generated by the gateway need to be simulated, if they can delay the transmission of a routed message. Usually the ECUs of the networks connected by the router/gateway run asynchronous, so the simulation should support varying the start times for the frame transmissions.

For the routing the simulation needs to have a model of the network controllers, with there internal buffers. A model of the implementation of the routing protocol and the buffering strategies. A model of the schedulable entities implementing the routing and other schedulable entities running on the gateway, that can impact the scheduling of

the routing related schedulable entities.

Advantages of the simulation are, that time of events can be determined at any point in the model. If the simulation is very performant, it can run faster than real-time, which makes it easier to observe longer time periods. The quality of results depend on the quality of the model. It is very challenging to create a model for the accurate generation of sporadic event messages. It may also be challenging to get an accurate model of a network controller, since they are intellectual property of the controller supplier and he may not be willing to share all required implementation details.

9.5.6.6.3 Measurement

There are some challenges when measuring the routing latency. For a router without any trace access, timing event can only be recorded on the source- and target network. The recording needs to be done in a way that allows to synchronize the traces from the source- and target network. To avoid faults when measuring the routing latency, the target network should be idle during the measurement. Otherwise routing latency may include the time the router/gateway is waiting for the network to allow the transmission. This time is not part of the routing latency. It is part of the transmission time of the message on the target network.

For a gateway with trace access, the routing latency can be measured on the gateway without the difficulty to synchronize traces from different networks. But this method comes with other downsides. Depending on the trace capabilities the measurements may impact the timing behavior. Usually the exact event time of reception and ready for transmission cannot be measured. For reception the measurement may only be possible on interrupt notification of the reception, missing the processing time of message by the network controller and the time until the interrupt gets asserted by the CPU. For the transmission the latest possible measurement will be after the handover of the message to the network controller, missing the time the message is processed by the controller and may be waiting in the internal buffer of the network controller, before it is ready for transmission. For certain use cases, it may be possible to neglect these time values, since they are small compared to the overall routing latency. Or a time delta can be added to the routing response time to compensate for the measurement error.

9.5.7 SPECIFIC METHOD Determine Response Time (CAN)

9.5.7.1 Scope and Application

Brief Description	The method yields the response time of a frame when transmitted on a CAN bus.
Reasoning	The method supports the determination of the resource needs of a CAN bus.

Table 9.57: Scope and Application

9.5.7.2 Classification

System	Network
Applied Protocol	CAN
Approach	Analysis / Simulation / Measurement

Table 9.58: Classification

9.5.7.3 Relation

Requirements	Interface input, see Table 9.60.
Process Steps	The method shall be applied during the following process steps: • Verification of data definition and of the configuration of the CAN bus • Requirement analysis for further development • Resource optimization during development phase
(Pre) Property	Depending on implementation this method requires the timing properties transmission time of all frames on the considered CAN bus (SPECIFIC PROPERTY Transmission Time (CAN).)
Belonging Post Property	GENERIC PROPERTY Latency

Table 9.59: Relation

9.5.7.4 Interface

The specific method Determine Response Time (CAN) is an instance of the GENERIC METHOD Determine Latency.

Input	The method requires parameters such as defined in 9.6, 9.7 and 9.8, i.e.: • CAN-ID (i.e. priority) of all CAN frames • The length / transmission time of all frames, their activation pattern (e.g. periodic/cyclic, sporadic) and stuff-bits • Execution speed of the CAN bus • Model of the spontaneous occurrence of event triggered frames
Boundary condition, Settings and Variants, Precondition	• Environmental states (like driving states)
Output	The result of this method is the response time of an individual frame on a CAN bus captured by the SPECIFIC PROPERTY Response Time (CAN).

Table 9.60: Interface

9.5.7.5 Limitation in Application

At the moment, there is no established treatment for the spontaneous occurrence of event-triggered frames. Therefore, a unified model or activation pattern for the spontaneous occurrence has to be applied in order to achieve comparable results between different configurations and between the applied approaches i.e. analysis/simulation/measurement.

9.5.7.6 Implementation

The implementation of the method for deriving the response time for CAN depends on the considered approach, namely analysis, simulation or measurement.

9.5.7.6.1 Analysis

The method for deriving response times for a CAN bus by analysis is based on mathematical formulas. These formulas can be implemented in a tool which supports the import of input parameters, the calculation of the individual frame response times and the export of the results.

The method has to enable the calculation of optimistic (best-case), average and pessimistic (worst-case) response time values. For that purpose, different assumptions regarding the number of stuff-bits shall be implemented, i.e. a minimum number (for the optimistic approach), an average number, and a maximum number (for the pessimistic approach). Furthermore, different models of the event-triggered frame activation patterns shall be supported. The derivation of the response times by analysis takes into the consideration the cyclic events with their periods and the spontaneous events with an event model. For example, the spontaneous events can be modeled with their debounce times as a "cycle" or with their maximum occurrence rate. Depending of the pessimistic or optimistic approach the calculation can estimate the upper bound with the lower limit of the period or with a specified period for the latter one.

Typical model elementes required for deriving response times for CAN buses by analysis are: (i) the underlying scheduling policy, for CAN buses this being SPNP (Static Priority Non-Preemptive), (ii) for each CAN frame, the priority given by the CAN frame identifier, the frame length (see 9.6 , 9.7) and the activation pattern (see 9.9) and (iii) the CAN bus speed (e.g. 100kBaud)(see table 9.7).

Based on these elements a formal analysis method, as for example presented in [19], computes response times for frames transmitted on CAN buses.

9.5.7.6.2 Simulation

Every frame will simulate with its activation (periodic, event triggered or mixed). In the simulation all frames try to access the network at their trigger points in time. If the simulated network is occupied by another frame the frame in question is delayed at least as long the virtual occupation lasts. Further, the blocking time is so long as the frame in question has a lower priority than all other frames tried to transmit at the same time. The response time is given by the difference of the point in time of the completed transmission and of the point in time of the send request.

9.5.7.6.3 Measurement

The measurement of the response time of the individual frame is only possible if the actual point in time for the send request is known. Thus, a correlated measurement with a common time base of the internal processes inside the ECU and on the network is necessary. One gets a distribution of the response time for each frame.

9.5.7.6.4 General remarks

The analysis, the simulation and the measurement should be implemented in a similar way. All boundary condition shall be revealed. Different algorithms can be applied as long as the results are identical under identical conditions. Any approximation shall be signalized and the parameter for the cut-off shall be open.

The frames shall be implemented with different deviation from the specified period in case of cyclic activation. Different possibilities for modeling event-triggered activation patterns shall be supported. For the response time analysis, one has to take into consideration the cyclic events with their periods and the spontaneous events with an event model. E.g. the spontaneous events can be modeled with their debounce times as a "cycle" or with their maximum occurrence rate. Depending of the pessimistic or optimistic approach the calculation of the response time can estimate the upper bound with the lower limit of the period or with a specified period for the latter one.

9.5.7.7 Determination of the Comparability of the Different Methods

Comparing analysis on one hand and simulation/measurement on the other hand the values of simulation/measurement for the response time shall be approach the analysis results in the long-time limit (under identical boundary conditions). Depending on the analysis method the difference should be small. In general, the simulation and the observation yield an optimistic approximation in the same manner depending on the sample/probe size (measurement/simulation time).

In order to compare the results of different methods (especially simulation/analysis and measurement) a check that all frames are contained in the output is highly recommended.

10 Artifacts for Timing Analysis

This chapter gives an overview of the artefacts (e.g. timing tasks, work products) from the use-cases. Additionally common elements for a timing model and timing-related work products are described.

10.1 Description of Timing Tasks

This section introduces timing tasks that have to be performed in order to accomplish the use-cases described in chapter 4, 5, 6 and 7. Timing tasks are generic descriptions of typical operations that can be performed with different scope (i.e. an entire ECU or an individual software component).

Activity	**Collect Timing Requirements**
Brief Description	Collect Timing Requirements
Description	Collect the known timing requirement from the specification documents. If necessary, find timing requirements in discussions/interviews with function owners and system architects. Derive new timing requirements from traces, measurements and experiments.
Relation Type	***Related Element***
Consumes	Function specification document, timing analysis report (optional)
Performed by	Timing engineer, function engineer and system engineer
Produces	Timing Requirements Document (TIMEX Extract) (see table 10.12)

Table 10.1: Task "Collect Timing Requirements"

Activity	**Compose**
Brief Description	Compose fine-graind elements into more coarse-grained structures.
Description	Compose lower level architecture elements (e.g. function) into a higher level architecture element (e.g. component) or timing property values of individual segments into a timing property value for the complete chain.
Relation Type	***Related Element***
Consumes	Low-level architecture elements, Timing Analysis Report
Performed by	E/E architect, function architect, software architect, timing engineer
Produces	High level architecture element, Timing Analysis Report

Table 10.2: Task "Compose"

Activity	**Create Implementation**
Brief Description	Create implementation of actual system
Description	The system is implemented according to the specification. If an existing implementation is available, this task can also be a manipulation or extension of the existing implementation. In both cases available timing requirements should be considered as soon as possible.
Relation Type	***Related Element***
Consumes	Existing implementation (optional), timing requirements
Performed by	Timing engineer, network engineer, system engineer
Produces	Implementation

Table 10.3: Task "Create Implementation"

Activity	**Create Timing Model**
Brief Description	Create analyzable timing-model
Description	The model parameters are derived from timing-related information. Incomplete information is estimated or generated based on synthesis-rules. Additional assumptions/operation scenarios/boundary conditions are documented. If an existing timing model is available, this task can also be a manipulation or extension of the existing timing model. In a model based development environment, the creation of a timing model can be used to elaborate different realization alternatives before taking the effort of implementation. In this case, timing requirements should be considered if possible.
Relation Type	***Related Element***
Consumes	Existing timing model (optional), timing requirements (optional)
Performed by	Timing engineer, network engineer, system engineer
Produces	Timing Model (see table 10.12)

Table 10.4: Task "Create Timing Model"

Activity	**Decompose**
Brief Description	Decompose a higher level architecture element into lower level architecture elements
Description	Decompose a higher level architecture element (e.g. feature) into lower level architecture elements (e.g. functions).
Relation Type	***Related Element***
Consumes	High level architecture element
Performed by	E/E architect, function architect, software architect
Produces	Low-level architecture elements

Table 10.5: Task "Decompose"

Activity	**Define Timing**
Brief Description	Define timing parameters and requirements
Description	In this task, the relevant timing parameters (e.g. latency) are identified for the considered level (e.g. feature) and requirement are defined for those.
Relation Type	***Related Element***
Consumes	Architecture element
Performed by	Timing engineer, network engineer, function engineer, function architect
Produces	Timing Requirement (see table 10.12)

Table 10.6: Task "Define Timing"

Activity	**Derive Timing Properties**
Brief Description	Derive timing properties by analyzing an existing implementation.
Description	Timing properties can be derived from a Timing Analysis Report from an existing implementation or for some messages directly from the communication matrix.
Relation Type	***Related Element***
Predecessor	Perform Implementation-Based Timing Analysis
Consumes	Timing Analysis Report (see table 10.12) or Communication Matrix (see table 10.12)
Performed by	Software architect, Network data engineer
Produces	ECU Timing (see table 10.12) or Timing Requirements (see table 10.12)

Table 10.7: Task "Derive Timing Properties"

Activity	**Map**
Brief Description	Map an element to component
Description	Map an element (like a function or a task) to a hardware or software component.
Relation Type	***Related Element***
Consumes	Element
Performed by	Software architect, E/E architect, Function architect
Produces	Mapping

Table 10.8: Task "Map"

Activity	**Perform Model-Based Timing Analysis**
Brief Description	Derive timing properties by performing timing analysis of the timing model
Description	Based on the timing model, analyze the timing (e.g. by simulation or static analysis; see Analytical calculation on page 178) and derive the timing properties.
Relation Type	***Related Element***
Predecessor	Create Timing Model
Consumes	Timing model
Performed by	Timing engineer or test engineer
Produces	Timing Analysis Report (see table 10.12) with timing properties according to Definition and Classification of Timing Properties on page 162

Table 10.9: Task "Perform Model-Based Timing Analysis"

Activity	**Perform Implementation-Based Timing Analysis**
Brief Description	Gain timing properties by observing the actual implementation
Description	Set up the environment (e.g. HIL or car) for the device to be analyzed. As the set of test conditions (stimulation model) can strongly influence the timing behavior, it is an essential part of the test environment and it has to be described precisely if reproducibility is required. Measure/trace the observable events and derive the timing properties. See Measurement and Tracing on page 182.
Relation Type	***Related Element***
Predecessor	Create Implementation
Consumes	Implementation, set of test conditions (stimulation model)
Performed by	Timing engineer or test engineer
Produces	Timing Analysis Report (see table 10.12) with timing properties according to Definition and Classification of Timing Properties on page 162

Table 10.10: Task "Perform Implementation-Based Timing Analysis"

Activity	**Verify Timing**
Brief Description	Verify adherence of timing requirements against timing properties
Description	Compare the timing requirements to the results of the timing analysis (timing properties). Generate a report that documents which timing requirements are fulfilled and which not.
Relation Type	***Related Element***
Predecessor	Collect Timing Requirements, Perform Timing Analysis
Consumes	Timing Requirements Document (TIMEX Extract) (see table 10.12),Timing Analysis Report (see table 10.12)
Performed by	Timing engineer
Produces	Timing Verification Report (see table 10.12)

Table 10.11: Task “Verify Timing”

10.2 Timing Model Elements

This section gives an overview of typical information necessary for a timing model. In general the type of information depends on the target architecture and different model elements, e.g. an ECU needs different artefacts to be configurable. The following graphic shows common artefacts on ECU and network level.

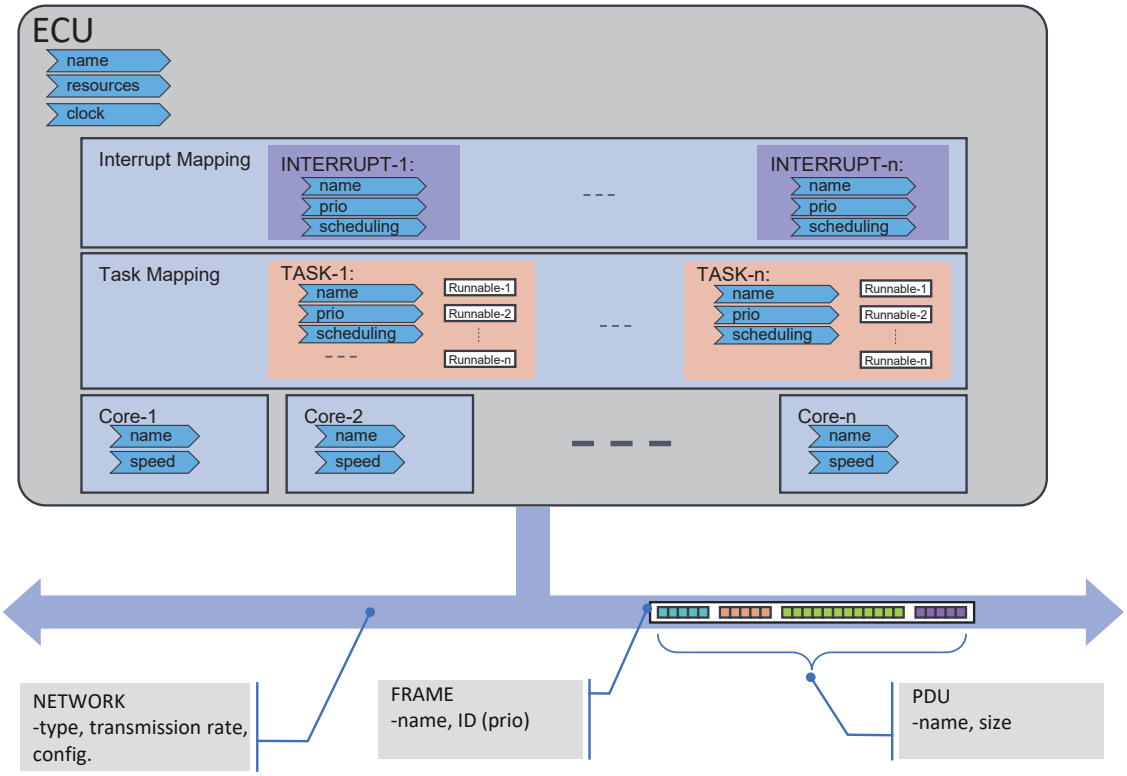

Figure 10.1: Overview of common elements relevant for a timing model

10.3 Work Products

This section introduces timing-related work products that are the outcome of or input to the use-cases described in chapter 4, 5, 6 and 7.

Name of Work Product	Description

Communication Matrix	The Communication Matrix describes the sender and receiver of messages exchanged on a network and the properties of these messages. E.g. the length of the messages, the interpretation of the message data, the cycle time of a message, debounce time of messages or number of message repetitions.
Chain	A chain defines a functional dependency between a start point(input) and an end point(output) of a function. If a function is implemented as a distributed function, the chain is composed of segments in the form of sub functions and the communication links connecting these sub function.
Feature	Feature is a customer vehicle function
Functional Architecture	A model representation of the set of interconnected function blocks (function network). It is a result of the feature decomposition.
Functional Requirements	A functional requirement is a specification of a behavior of a system.
ECU Extract	A document that describes the ECU specific view on the System Description. Among other information it contains the Atomic Software Components mapped to the ECU and a description of the network interfaces of the ECU.
ECU Timing	TimingDescription and TimingConstraints defined for a concrete ECU taking the ECU configuration and the ECU Software Composition (including their implementation) into account.
ECU Configuration Values	They are a collection of all configuration values for an ECU. This includes the configuration settings of the RTE with the runnable to task mapping and the configuration settings of the OS with the description of tasks and interrupts.
Execution Manifest	The execution manifest defines the process with all its properties. It is defined for a specific machine by referencing its modes in the startup configuration. One execution manifest is defined per process.
Expert Knowledge	In early stages of the development when measurements are not available yet, still decision have to be made to progress the development. These decisions are made based on estimates or best guesses from an experienced engineer. In the SPEM diagrams this is represented by this work product *Expert Knowledge*.
E/E Architecture Model	A model representation of the networks, gateways and ECUs
Implementation	A realization of the specification.

Machine Manifest	Description of deployment content for the configuration of the machine, independent of any service instances or applications.
Mapping	Assignment of an element (like a function or a task) to a hardware or software component.
Segment	A segment is a part of a chain. A segment can be a sub function mapped onto a computation resource or a communication link connecting two sub functions.
Set of Test Conditions	Test parameters generally used to provoke a failure
Timing Model	A model representation of the system that is sufficiently complete to analyze the timing properties of the system e.g. through simulation or formal methods.
Timing Analysis Report	A document summarizing the timing properties of the system. It can consolidate both measurement-based as well as model-based timing properties.
Timing Requirements Document (TIMEX Extract)	A document containing an explicit set of timing-related requirements. This document may be included e.g. in a specification document handed from an OEM or Tier-1 to a supplier. The document includes: **a)** timing-requirements related to the functionality (e.g. reaction time to driver action) **b)** timing-requirements related to the platform (e.g. load constraints for all ECUs to meet safety margins)
Timing Verification Report	A document that contains an overview over all timing requirements contained in the system and in how far they are fulfilled by the current implementation.

Table 10.12: Timing-related Work Products

11 Limitations

No limitations in this release.

A Timing Reference Platform

A.1 Introduction

The AUTOSAR AP Demonstrator does not consider timing aspects and there is no publicly available CP part, so a Timing Reference Platform (TRP) is built. It is required to gather experiences with unified timing and tracing on AP and CP and to validate the newly developed methods.

A.2 Relation and Extensions to general AUTOSAR Demonstrator

The TRP is based on AUTOSAR Demonstrator R23-11 (AP demonstrator) and uses the OS and FC components (esp. ara::com, ara::log). Because none of the provided applications is optimal for timing analyses, the TRP uses an own dedicated application. The TRP is built in the same way as the AP demonstrator is built, so the build process is not described in this document. If future releases of the AP demonstrator provide functions that are relevant for timing and tracing, the TRP will be upgraded to those releases.

The reference application consists of an AP part that subscribes to a service and a second part that provides the required service. The service provider application can be implemented on AP and CP based ECUs.

Setup with both parts implemented on AP:

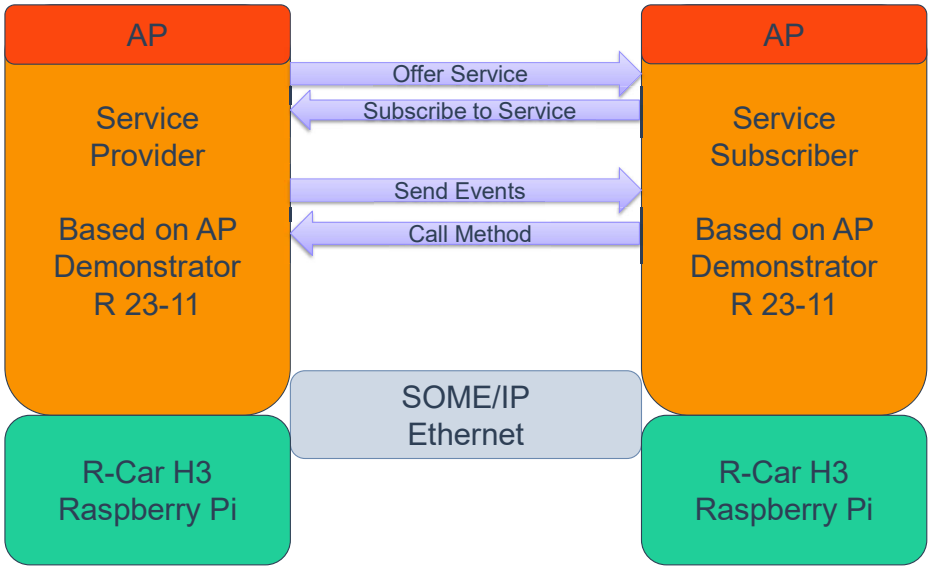

Figure A.1: Setup AP - AP

Setup with one part on AP and one part on CP:

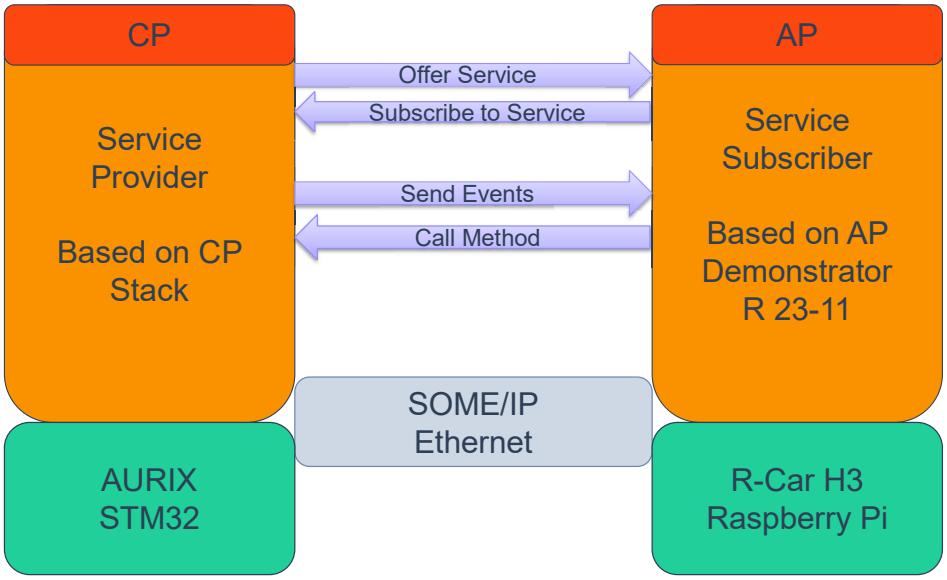

Figure A.2: Setup AP - CP

A.3 Design of TRP on Functional Level

The application of TRP is designed on functional level to demonstrate, how timing can be considered in this early stage of the development process. A simple sensor/actuator system is used with an end-to-end timing constraint 150ms. Complex processing between sensor and actuator is required, but as the focus is on timing and not not functionality, details of processing are not considered. As shown in figure A.3 the processing is decomposed into multiple blocks and the end-to-end timing requirement is broken down accordingly: If a budget of 10ms for sensor and and 15ms for actuator is given, the remaining 125ms used as budgets for decomposed processing functions. Additionally a frequency of 100Hz for the sensor data is specified.

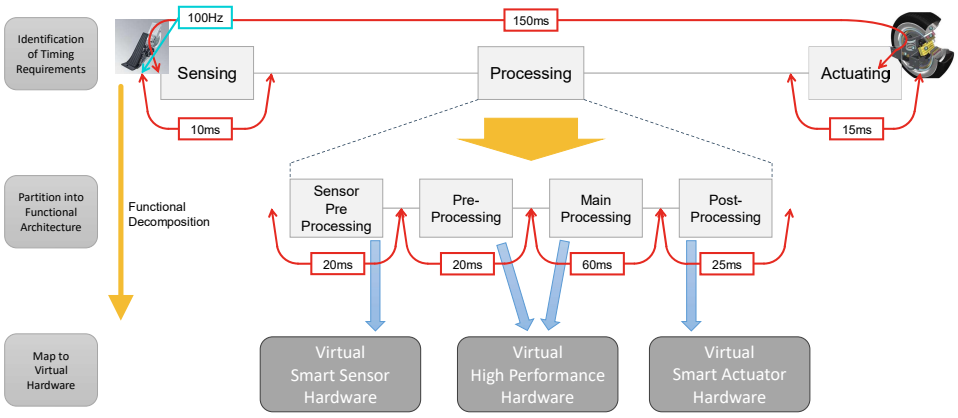

Figure A.3: Design of TRP on Functional Level

Before the software components for CP and AP can be designed and implemented, the blocks from functional level are transformed to abstract platform and the budgets are broken down fit the new structure of abstract software components. Additionally the budgets are decomposed into processing time and communication time. The result is shown figure A.4.

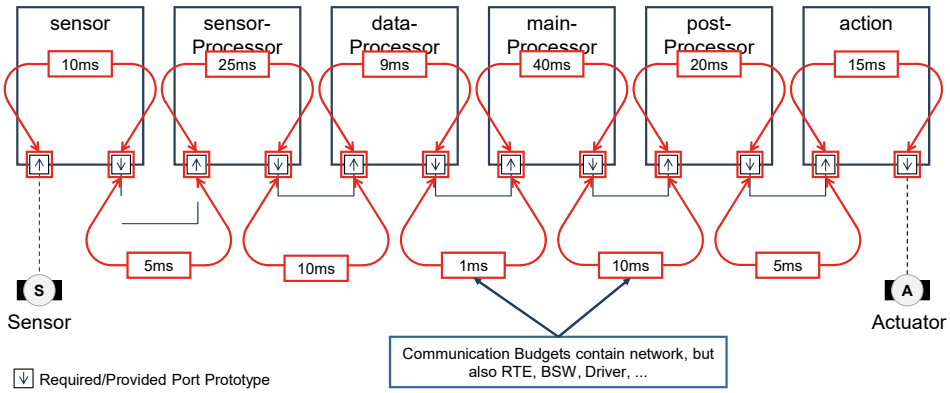

Figure A.4: Design of TRP for Abstract Platform

The components from XP are transformed into software components for CP and AP. The budgets from XP are inherited by the software components and can be further decomposed together with the components. If the basic software is modeled in details, especially on CP, the communication budgets from XP have to be decomposed into budgets for BSW components and network.

A.4 Software Application of TRP

The software for TRP consists of an AdaptiveApplicationSoftwareComponent hpProcessingApp that subscribes to a service. It implements the blocks DataProcessor and MainProcessor. The service can be provided either by a classic SWC SmartSensorActuator, that implements the blocks SensorProcessor and PostProcessor, or by an other A-SWC. The classic SWC is used to build a distributed system with one AP ECU and one CP ECU.

The service interface consists of an Event that provides the sensor data as uint_32 and a method that can be called to invoke an action at the provider. The sensor data is incremented periodically (internally by using an internal abstract sensor) and reset to 0 after reaching a value of 26. If the sensor data is greater than 25, the method action from service interface is called by hpProcessingApp to trigger the actuator. The functionality is intentionally kept simple, because the focus is on timing.

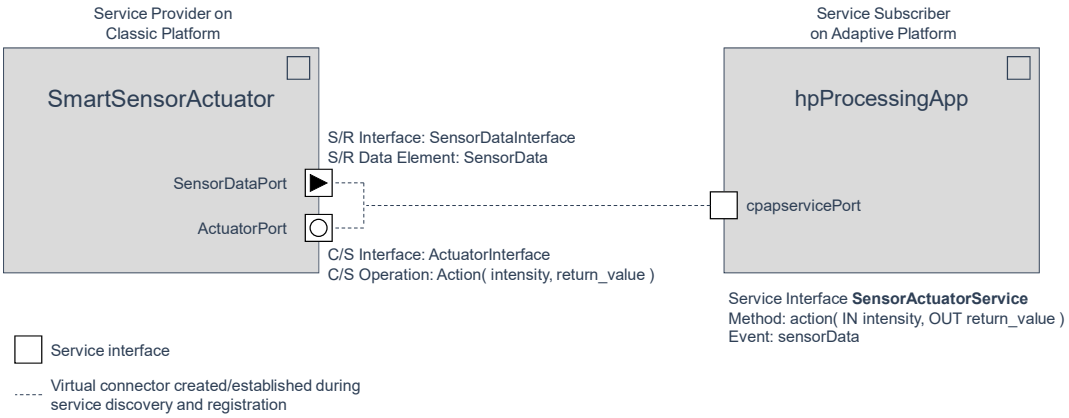

Figure A.5: VFB View on CP - AP variant

A.4.1 Software Application of AP Subscriber

The application HpProcessingApp comprises implicitly and explicitly created threads. The complex internal structure is needed to show different types of thread creation and activation. When the application is started, it starts service discovery and registers a callback, that is called when the service is available. The generator of AUTOSAR demonstrator is used to generate C++ code for service interface and for network binding to SOME/IP.

A callback(`cb_wegres_sensor`) is registered for the event with sensor data. The callback is executed in a dedicated, implicitly created thread, when new sensor data is received. This is managed by ara::com in a way that is transparent for the application. For each received event 8 worker threads are created(`extra_worker_X`) and the processed data is prepared next step after all worker threads have finished.

A periodic thread (`wgres_periodic`) checks for availability of new data. In case of new data, it activates thread (`wgres_trigger`), that finally calls the action method.

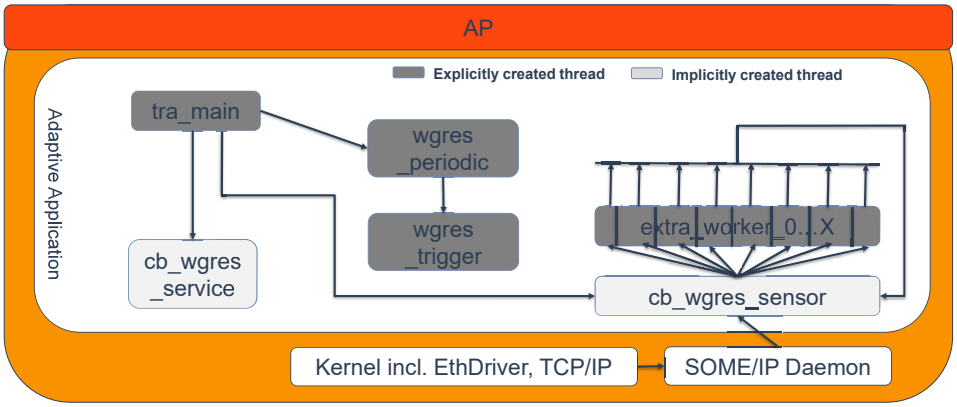

Figure A.6: Internal strucure of AP subscriber

A.4.2 Software Application of CP Provider

As there is no public AUTOSAR demonstrator for the classic platform, the implementation of service provider for CP is vendor specific. ARXML files with SWCs are provided; the contained SWC RootComposition can be used to implement the SWC DataPreprocessor from above.

The service provider consists of two SWCs: `SensorProcessor` provides the sensor data via a sender/receiver port and `PostProcessor` with a client/server port to process data and trigger final (brake) action. Both ports have to be made avaliable as a SOME/IP service e.g. with a SOME/IP-Transformer.

SWC `SensorProcessor` is composed of two SWCs: `Filter` reads and filters the sensor data and `Adjustment` sends the sensor data periodically after adjustment. The RTE Events that activate the Runnalbes are not shown here.

SWC `PostProcessor` directly implements the action in three Runnalbes. Depending on the implementation stack and hardware the performed action can be visualized (e.g. LED on evaluation board).

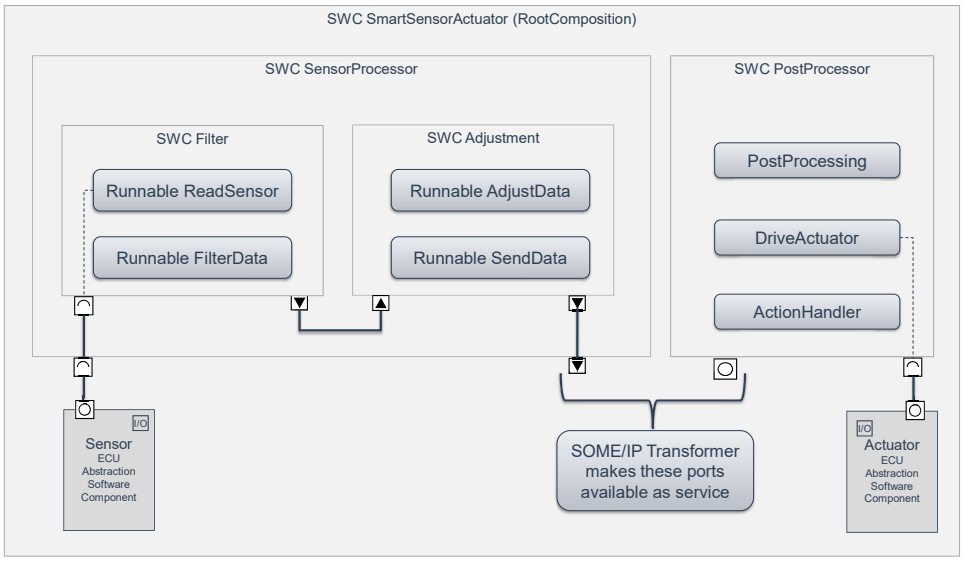

Figure A.7: Internal strucure of CP provider

A.4.3 Software Application of AP Provider

The implementation of service provider for AP is very simple and straight forward: The service is offered immediately at startup. When sending of sensor data is started a new thread is created that sends the incremented sensor data and sleeps for 250ms before sending next value. If the action method is called, a message is written to logger.

A.5 Hardware Setup of TRP

As the TRP is based on the AP demonstrator the hardware support is the same for AP part (tested with Renesas H3 and Rasperry Pi 3). For CP part different commercial

solutions were used, so HW support depends on vendor. Additionally an Ethernet switch is necessary and a monitoring port is highly recommended.

A.6 Tracing and Measurement on TRP

A.6.1 Tracing and Measurement on AP

The service subscriber application uses the ara::log tracing API to insert user based trace events. The ara::com FC implements trace events of the ara::log API for subscribing to the service and for sending and receiving messages of the service. On OS level, Linux lttng hooks are used to call ARTI hooks.

A.6.2 Tracing and Measurement on CP

On CP, VFB Trace and ARTI Hooks are used to trace the application. On OS level, either vendor specific hooks are mapped to ARTI hooks, or the ARTI hooks are written directly into the OS code. The OS hooks trace task switches and task state changes. On VFB level, the RTE VFB Tracing Hooks are mapped to according ARTI hooks.

A.7 Evaluation of requirements on TRP

In order to guide the development and integration regarding temporal characteristics of components and systems, the Timing Reference Platform (TRP) highlights the capabilities of specifying timing requirements imposed on a system and its components. This enables the conduct of timing analysis and exploration as well as the comparison of results against timing requirements. The necessary tools to do so are specified within the document AUTOSAR_TPS_TimingExtensions. In order to evaluate end-2-end requirements on a distributed system, an event chain has to be constructed. The following example is focused on the section starting at the availability of sensor data after preprocessing (Runnable SendData) on CP part and ending at the reaction in the actuator (Runnalbe ActionHandler). For the whole event chains a timing requirement of 70ms is applied.

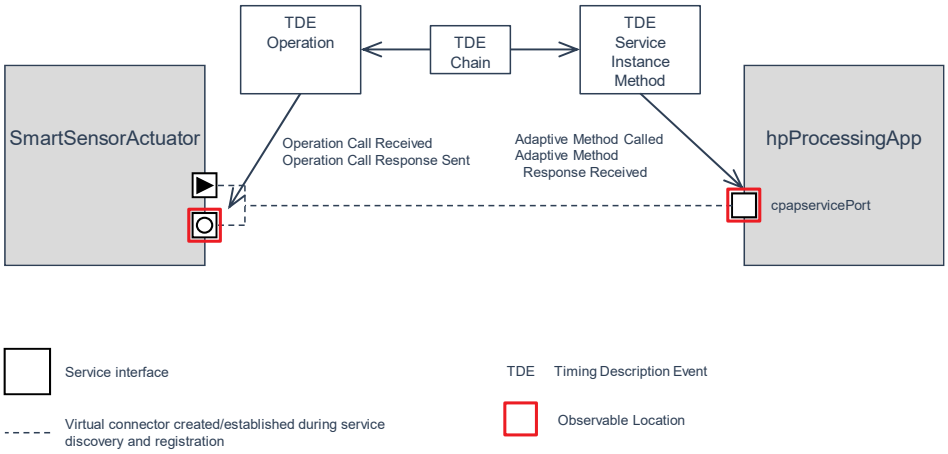

Figure A.8: Event chain covering an CP - AP communication

Starting point are events, which specify observable locations within the system at which something happens, i.e., an internal state is changed. As depicted in Figure A.8, in case of the Timing Reference Platform (TRP), such events are, for example, Operation Call Received, Operation Call Response Sent on the Classic Plattform and Adaptive Method Called and Adaptive Method Response Received on the Adaptive Platform. With the help of an event chain, then, a cause-effect relationship between the Classic and the Adaptive Platform is created. In other words, a port access on the one platform plays the role of a stimulus event and another port access on the other platform plays the role of a response event. Finally, this event chain can be referenced by a timing constraint, e.g., Latency Timing Constraint that allows one to make a specific timing demand which can be measured and evaluated on the actual system. For reaction on sensor data a Latency Timing Constraint of 70ms is used (see Figure A.9).

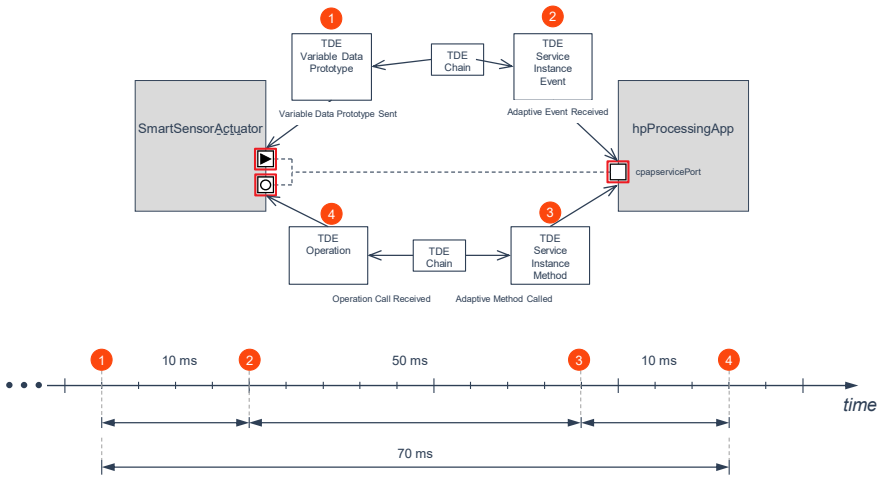

Figure A.9: End-2-end Event Chain with Segments

The cross-platform communication might not only imply cross-core but also network communication and the processing on the CP part takes some time. As a consequence, execution time constraints one the Runnables are required. For example a minimum (100us) and maximum (500us) time boundary for the net execution of Runnalbe ActionHandler entity is added.

The tracing itself as well as the trace analysis process is out of scope of AUTOSAR. There are various commercial and non-commercial tools available that are able to perform the necessary tracing and analysis. Ideally, the related ASAM ARTI standard should be used for tracing. However, the output of the analysis shall serve as input for the validation against the AUTOSAR timing constraints. In case of the TRP, the measurements shall evaluate the timings defined in the TIMEX constraints and show a valid timing.

B TIMEX ARTI Mapping

B.1 Introduction

TIMEX specifies timing behavior and timing constraints on specific events that happen in the AUTOSAR application. The so called Timing Description Events of TIMEX (TDEs) are defined on the system model level. In order to verify the requirements set with TIMEX, the actual system behavior needs to be measured. Measuring the timing behavior is usually done by collecting the events in a trace with timestamps with a consecutive analysis of the times elapsed between the events. To trace the events, the application has to implement so called trace points, where the desired event is written to the trace.

As TDEs are defined on system level, they are not directly related to the executed code. Depending on the further detailing of the model, the actual implementation may look different on various systems. As a consequence, the existing tracing hooks of AUTOSAR (VFB tracing, ara::log tracing, ARTI), which focus on implementation level, cannot be mapped easily to the TIMEX events.

This chapter is a guideline how Timing Description Events can be mapped to implementation level tracing mechanisms. It is not normative, only explanatory. It is not exhaustive, especially as some TDEs are hard to map to any code in the application, or are implementation specific in a way that there is no guidance possible.

B.2 Mapping on AUTOSAR Classic Platform

The following tables show how Timing Description Events of the AUTOSAR Classic Platform map to VFB Trace events of the RTE and further to ARTI tracing hooks, if available.

TIMEX Event	**TDEventVariableDataPrototype**
Event Type	**VariableDataPrototypeReceived**
Code Location	invocation/return of `Rte_Read_<port>_<vdp>(<data>)` invocation/return of `Rte_Dread_<port>_<vdp>()` invocation/return of `Rte_Iread_<re>_<port>_<vdp>()` invocation/return of `Rte_Receive_<port>_<vdp>(<data)`

VFB Trace Event	`Rte_Arti_ReadHook_<cts>_<port>_<vdp>_Start/Return(<data>)`
	`Rte_Arti_DreadHook_<cts>_<port>_<vdp>_Start/Return()`
	`Rte_Arti_IreadHook_<cts>_<re>_<port>_<vdp>_Start/Return()`
	`Rte_Arti_ReceiveHook_<cts>_<port>_<vdp>_Start/Return(<data>)`

ARTI	`ARTI_TRACE_N(USER, ARTI_CP_RTE_API, shortNameOf(<cts>), <instance_ptr|0>, Read_Start/Return, numberOf(<data>)+1, idOf(<ap>), <data>)`
	`ARTI_TRACE_N(USER, ARTI_CP_RTE_API, shortNameOf(<cts>), <instance_ptr|0>, DRead_Start/Return, numberOf(<data>)+1, idOf(<ap>), <data>)`
	`ARTI_TRACE_N(USER, ARTI_CP_RTE_API, shortNameOf(<cts>), <instance_ptr|0>, IRead_Start/Return, numberOf(<data>)+1, idOf(<ap>), <data>)`
	`ARTI_TRACE_N(USER, ARTI_CP_RTE_API, shortNameOf(<cts>), <instance_ptr|0>, Receive_Start/Return, numberOf(<data>)+1, idOf(<ap>), <data>)`

Remarks	Depending on the use case, the `Start` or `Return` VFB Trace Hook should be used.

Table B.1: Mapping of VariableDataPrototypeReceived

TIMEX Event	**TDEventVariableDataPrototype**
Event Type	**VariableDataPrototypeSent**
Code Location	invocation/return of `Rte_Write_<port>_<vdp>(<data>)` invocation/return of `Rte_Iwrite_<re>_<port>_<vdp>()` invocation/return of `Rte_Send_<port>_<vdp>(<data)`

VFB Trace Event	`Rte_Arti_WriteHook_<cts>_<port>_<vdp>_Start/Return(<data>)`
	`Rte_Arti_IwriteHook_<cts>_<re>_<port>_<vdp>_Start/Return()`
	`Rte_Arti_SendHook_<cts>_<port>_<vdp>_Start/Return(<data>)`

ARTI	`ARTI_TRACE_N(USER, ARTI_CP_RTE_API, shortNameOf(<cts>), <instance_ptr|0>, Write_Start/Return, numberOf(<data>)+1, idOf(<ap>), <data>)`
	`ARTI_TRACE_N(USER, ARTI_CP_RTE_API, shortNameOf(<cts>), <instance_ptr|0>, IWrite_Start/Return, numberOf(<data>)+1, idOf(<ap>), <data>)`
	`ARTI_TRACE_N(USER, ARTI_CP_RTE_API, shortNameOf(<cts>), <instance_ptr|0>, Send_Start/Return, numberOf(<data>)+1, idOf(<ap>), <data>)`

Remarks	Depending on the use case, the `Start` or `Return` VFB Trace Hook should be used.

Table B.2: Mapping of VariableDataPrototypeSent

TIMEX Event	**TDEventOperation**
Event Type	**OperationCalled**
Code Location	invocation of `Rte_Call_<port>_<op>(<data>)`
VFB Trace Event	`Rte_Arti_CallHook_<cts>_<port>_<op>_Start(<data>)`
ARTI	`ARTI_TRACE_N(USER, ARTI_CP_RTE_API, shortNameOf(<cts>), <instance_ptr|0>, Call_Start, numberOf(<data>)+1, idOf(<ap>), <data>)`

Table B.3: Mapping of OperationCalled

TIMEX Event	**TDEventOperation**
Event Type	**OperationCallResponseReceived**

Code Location	synchronous: return of `Rte_Call_<port>_<op>(<data>)` asynchronous: return of `Rte_Result_<port>_<op>(<data>)`

VFB Trace Event	synchronous:
	`Rte_Arti_CallHook_<cts>_<port>_<op>_Return(<data>)`
	asynchronous:
	`Rte_Arti_ResultHook_<cts>_<port>_<op>_Return(<data>)`

ARTI	synchronous:
	`ARTI_TRACE_N(USER, ARTI_CP_RTE_API, shortNameOf(<cts>), <instance_ptr|0>, Call_Return, numberOf(<data>)+1, idOf(<ap>), <data>)`
	aynchronous:
	`ARTI_TRACE_N(USER, ARTI_CP_RTE_API, shortNameOf(<cts>), <instance_ptr|0>, Response_Return, numberOf(<data>)+1, idOf(<ap>), <data>)`

Remarks	In case of asynchronous calls, all `Rte_Response` events should be traced to ARTI. Only the ones returning `RTE_E_OK` shall be mapped to the TIMEX event.

Table B.4: Mapping of OperationCallResponseReceived

TIMEX Event	**TDEventOperation**
Event Type	**OperationCallReceived**
Code Location	invocation of `<op>(<data>)` (see SWS_RTE, OperationInvokedEvent)
VFB Trace Event	`Rte_Arti_Runnable_<cts>_<op>_Start()`
ARTI	`ARTI_TRACE(USER,AR_CP_RTE_RUNNABLE, shortNameOf(<cts>),0,RteRunnable_Start,idOf<op>)`

Table B.5: Mapping of OperationCallReceived

TIMEX Event	**TDEventOperation**
Event Type	**OperationCallResponseSent**
Code Location	return of `<op>(<data>)` (see SWS_RTE, OperationInvokedEvent)
VFB Trace Event	`Rte_Arti_Runnable_<cts>_<op>_Return()`
ARTI	`ARTI_TRACE(USER,AR_CP_RTE_RUNNABLE, shortNameOf(<cts>),0,RteRunnable_Return,idOf<op>)`

Table B.6: Mapping of OperationCallResponseSent

TIMEX Event	**TDEventModeDeclaration**
Event Type	**ModeDeclarationSwitchInitiated**
Code Location	invocation of `Rte_Switch_<port>_<md>(<mode>))`
VFB Trace Event	`Rte_Arti_SwitchHook_<cts>_<port>_<md>_Start(<mode>)`
ARTI	n/a

Table B.7: Mapping of ModeDeclarationSwitchInitiated

TIMEX Event	**TDEventModeDeclaration**
Event Type	**ModeDeclarationSwitchCompleted**

Code Location	invocation of `Rte_SwitchAck_<port>_<md>())`
VFB Trace Event	`Rte_Arti_SwitchAckHook_<cts>_<port>_<md>_Start()`
ARTI	n/a

Table B.8: Mapping of ModeDeclarationSwitchCompleted

TIMEX Event	**TDEventTrigger**
Event Type	**TriggerActivated**
Code Location	invocation of `<re>()` (see SWS_RTE, External/InternalTriggerOccurredEvent)
VFB Trace Event	`Rte_Arti_Runnable_<cts>_<re>_Start()`
ARTI	`ARTI_TRACE(USER,AR_CP_RTE_RUNNABLE,` `shortNameOf(<cts>),0,RteRunnable_Start,idOf(<re>))`

Table B.9: Mapping of TriggerActivated

TIMEX Event	**TDEventTrigger**
Event Type	**TriggerReleased**
Code Location	invocation of `Rte_Trigger_<port>_<trigger>()` invocation of `Rte_IrTrigger_<re>_<trigger>()`
VFB Trace Event	`Rte_Arti_TriggerHook_<cts>_<port>_<trigger>_Start()` `Rte_Arti_IrTriggerHook_<cts>_<re>_<trigger>_Start()`

ARTI
 	
ARTI
	`ARTI_TRACE_N(USER, AR_CP_RTE_API,`
	`shortNameOf(<cts>), <instance_ptr|0>, Trigger_Start,`
	`1, idOf(<port>_<trigger>))`
	`ARTI_TRACE_N(USER, AR_CP_RTE_API,`
	`shortNameOf(<cts>), <instance_ptr|0>, IrTrigger_Start,`
	`1, idOf(<re>_<trigger>))`

Table B.10: Mapping of TriggerReleased

TIMEX Event	**TDEventSwcInternalBehavior**
Event Type	**RunnableEntityActivated**
Code Location	(implementation specific)
VFB Trace Event	n/a
ARTI	n/a
Remark	The activation of runnables is very implementation specific and may not be observable/traceable.

Table B.11: Mapping of RunnableEntityActivated

TIMEX Event	**TDEventSwcInternalBehavior**
Event Type	**RunnableEntityStarted**
Code Location	invocation of `<re>(<data>))`
VFB Trace Event	`Rte_Arti_Runnable_<cts>_<re>_Start()`
ARTI	`ARTI_TRACE(USER,AR_CP_RTE_RUNNABLE,` `shortNameOf(<cts>),0,RteRunnable_Start,idOf<re>)`

Table B.12: Mapping of RunnableEntityStarted

TIMEX Event	TDEventSwcInternalBehavior
Event Type	**RunnableEntityTerminated**
Code Location	return of `<re>(<data>))`
VFB Trace Event	`Rte_Arti_Runnable_<cts>_<re>_Return()`
ARTI	`ARTI_TRACE(USER,AR_CP_RTE_RUNNABLE,` `shortNameOf(<cts>),0,RteRunnable_Return,idOf<re>)`

Table B.13: Mapping of RunnableEntityTerminated

TIMEX Event	TDEventSwcInternalBehavior
Event Type	**RunnableEntityVariableAccess**
Code Location	invocation/return of `Rte_Read_<port>_<vdp>(<data>)` invocation/return of `Rte_Dread_<port>_<vdp>()` invocation/return of `Rte_Iread_<re>_<port>_<vdp>()` invocation/return of `Rte_Receive_<port>_<vdp>(<data>)` invocation/return of `Rte_Write_<port>_<vdp>(<data>)` invocation/return of `Rte_Iwrite_<re>_<port>_<vdp>()` invocation/return of `Rte_Send_<port>_<vdp>(<data>)` invocation/return of `Rte_IrvRead_<re>_<irvdp>()` invocation/return of `Rte_IrvIread_<re>_<irvdp>()` invocation/return of `Rte_IrvWrite_<re>_<irvdp>(<data>)` invocation/return of `Rte_IrvIwrite_<re>_<irvdp>()`

VFB Trace Event	`Rte_Arti_ReadHook_<cts>_<port>_<vdp>_Start/Return(<data>)`
	`Rte_Arti_DreadHook_<cts>_<port>_<vdp>_Start/Return()`
	`Rte_Arti_IreadHook_<cts>_<re>_<port>_<vdp>_Start/Return()`
	`Rte_Arti_ReceiveHook_<cts>_<port>_<vdp>_Start/`
	`Return(<data>)`
	`Rte_Arti_WriteHook_<cts>_<port>_<vdp>_Start/Return(<data>)`
	`Rte_Arti_IwriteHook_<cts>_<re>_<port>_<vdp>_Start/Return()`
	`Rte_Arti_SendHook_<cts>_<port>_<vdp>_Start/Return(<data>)`
	`Rte_Arti_IrvReadHook_<cts>_<re>_<irvdp>_Start/Return()`
	`Rte_Arti_IrvIreadHook_<cts>_<re>_<irvdp>_Start/Return()`
	`Rte_Arti_IrvWriteHook_<cts>_<re>_<irvdp>_Start/`
	`Return(<data>)`
	`Rte_Arti_IrvIwriteHook_<cts>_<re>_<irvdp>_Start/Return()`

ARTI	`ARTI_TRACE_N(USER, AR_CP_RTE_API, shortNameOf(<cts>), <instance_ptr|0>, Read_Start/Return, numberOf(<data>)+1, idOf(<ap>), <data>)`
	`ARTI_TRACE_N(USER, AR_CP_RTE_API, shortNameOf(<cts>), <instance_ptr|0>, DRead_Start/Return, 1 , idOf(<ap>))`
	`ARTI_TRACE_N(USER, AR_CP_RTE_API, shortNameOf(<cts>), <instance_ptr|0>, IRead_Start/Return, 1 , idOf(<ap>))`
	`ARTI_TRACE_N(USER, AR_CP_RTE_API, shortNameOf(<cts>), <instance_ptr|0>, Receive_Start/Return, numberOf(<data>)+1 , idOf(<ap>), <data>)`
	`ARTI_TRACE_N(USER, AR_CP_RTE_API, shortNameOf(<cts>), <instance_ptr|0>, Write_Start/Return, numberOf(<data>)+1 , idOf(<ap>), <data>)`
	`ARTI_TRACE_N(USER, AR_CP_RTE_API, shortNameOf(<cts>), <instance_ptr|0>, IWrite_Start/Return, numberOf(<data>)+1 , idOf(<ap>), <data>)`
	`ARTI_TRACE_N(USER, AR_CP_RTE_API, shortNameOf(<cts>), <instance_ptr|0>, Send_Start/Return, numberOf(<data>)+1 , idOf(<ap>), <data>)`
	`ARTI_TRACE_N(USER, AR_CP_RTE_API, shortNameOf(<cts>), <instance_ptr|0>, IrvRead_Start/Return, numberOf(<data>)+1 , idOf(<ap>), <data>)`
	`ARTI_TRACE_N(USER, AR_CP_RTE_API, shortNameOf(<cts>), <instance_ptr|0>, IrvIRead_Start/Return, 1 , idOf(<ap>))`
	`ARTI_TRACE_N(USER, AR_CP_RTE_API, shortNameOf(<cts>), <instance_ptr|0>, IrvWrite_Start/Return, numberOf(<data>)+1 , idOf(<ap>), <data>)`
	`ARTI_TRACE_N(USER, AR_CP_RTE_API, shortNameOf(<cts>), <instance_ptr|0>, IrvIWrite_Start/Return, numberOf(<data>)+1 , idOf(<ap>), <data>)`

Remarks	Depending on the use case, the `Start` or `Return` VFB Trace Hook should be used.

Table B.14: Mapping of RunnableEntityVariableAccess

TIMEX Event	**TDEventBswInternalBehavior**
Event Type	**BswModuleEntityActivated**
Code Location	(implementation specific)
VFB Trace Event	n/a
ARTI	n/a
Remarks	The activation of a BswModuleEntity is very implementation specific and usually not observable/traceable.

Table B.15: Mapping of BswModuleEntityActivated

TIMEX Event	**TDEventBswInternalBehavior**
Event Type	**BswModuleEntityStarted**
Code Location	if BswSchedulableEntity: invocation of `<entity>()` if BswInterruptEntity(CAT1): invocation of CAT-1 interrupt if BswInterruptEntity(CAT2): invocation of CAT-2 ISR if BswCalledEntity: invocation of `<entity>(<data>)` called by SchM_Call()

VFB Trace Event	`SchM_Arti_Schedulable_<bsnp>_<entity>_Start()`
	n/a (no event for CAT-1 interrupt)
	n/a (no event for CAT-2 ISR)
	`SchM_Arti_CallHook_<bsnp>_<entity>_Start(<data>)`

ARTI	`ARTI_TRACE(USER,AR_CP_SCHM_SCHEDULABLE,`
	`<bsnp>,0,SchMSchedulable_Start,idOf<entity>)`
	`ARTI_TRACE(NOSUSP,AR_CP_ARTI_CAT1ISR,`
	`<os>,<core>,OsCat1Isr_Start,idOf<cat1isr>)`
	`ARTI_TRACE(NOSUSP,AR_CP_OS_CAT2ISR,`
	`<os>,<core>,OsCat2Isr_Start,idOf<cat2isr>)`
	n/a (no tracepoint for SchM called entity)

Table B.16: Mapping of BswModuleEntityStarted

TIMEX Event	**TDEventBswInternalBehavior**
Event Type	**BswModuleEntityTerminated**
Code Location	if BswSchedulableEntity: return of `<entity>()` if BswInterruptEntity(CAT1): return of CAT-1 interrupt if BswInterruptEntity(CAT2): return of CAT-2 ISR if BswCalledEntity: return of `<entity>(<data>)` called by SchM_Call()

VFB Trace Event	`SchM_Arti_Schedulable_<bsnp>_<entity>_Return()`
	n/a (no event for CAT-1 interrupt)
	n/a (no event for CAT-2 ISR)
	`SchM_Arti_CallHook_<bsnp>_<entity>_Return(<data>)`
	(only if synchronous!)

ARTI	`ARTI_TRACE(USER,AR_CP_SCHM_SCHEDULABLE,`
	`<bsnp>,0,SchMSchedulable_Return,idOf<entity>)`
	`ARTI_TRACE(NOSUSP,AR_CP_ARTI_CAT1ISR,`
	`<os>,<core>,OsCat1Isr_Stop,idOf<cat1isr>)`
	`ARTI_TRACE(NOSUSP,AR_CP_OS_CAT2ISR,`
	`<os>,<core>,OsCat2Isr_Stop,idOf<cat2isr>)`
	n/a (no tracepoint for SchM called entity)

Table B.17: Mapping of BswModuleEntityTerminated

TIMEX Event	**TDEventBswModule**
Event Type	**BswMEntryCalled**
Code Location	invocation of `<entity>()`
VFB Trace Event	n/a
ARTI	`ARTI_TRACE_N(USER, AR_CP_BSW_API, <mip>, 0, Bsw_Start,` `numberOf(param...)+1 , idOf(<service>), <param...>)`

Table B.18: Mapping of BswMEntryCalled

TIMEX Event	TDEventBswModule
Event Type	**BswMEntryCallReturned**
Code Location	return of `<entity>()`
VFB Trace Event	n/a
ARTI	`ARTI_TRACE_N(USER, AR_CP_BSW_API, <mip>, 0, Bsw_Return, numberOf(param...)+1 , idOf(<service>), <param...>)`

Table B.19: Mapping of BswMEntryCallReturned

TIMEX Event	TDEventBswModeDeclaration
Event Type	**ModeDeclarationRequested**
Code Location	Receiving Bsw_modeRequestPort
VFB Trace Event	n/a
ARTI	n/a

Table B.20: Mapping of ModeDeclarationRequested

TIMEX Event	TDEventBswModeDeclaration
Event Type	**ModeDeclarationSwitchInitiated**
Code Location	Sending Bsw_modeSwitchPort
VFB Trace Event	n/a
ARTI	n/a

Table B.21: Mapping of ModeDeclarationSwitchInitiated

TIMEX Event	TDEventBswModeDeclaration
Event Type	**ModeDeclarationSwitchCompleted**
Code Location	Switch done (OS internal)
VFB Trace Event	n/a
ARTI	n/a

Table B.22: Mapping of ModeDeclarationSwitchCompleted

TIMEX Event	TDEventISignal
Event Type	**ISignalSentToCOM**
Code Location	return from `Com_ReceiveSignal()`
VFB Trace Event	`Rte_Arti_ComHook_<signal>_SigTx(<data>)`
ARTI	n/a

Table B.23: Mapping of ISignalSentToCOM

TIMEX Event	TDEventISignal
Event Type	**ISignalAvailableForRTE**
Code Location	invocation of `Com_SendSignal()`
VFB Trace Event	`Rte_Arti_ComHook_<signal>_SigRx(<data>)`
ARTI	n/a

Table B.24: Mapping of ISignalAvailableForRTE

B.3 Mapping on AUTOSAR Adaptive Platform

The following tables show how Timing Description Events of the AUTOSAR Adaptive Platform map to ara::log trace events and further to ARTI trace events, if available. The code location in APD is an example referring to the timing reference platform as described in Appendix A.

TIMEX Event	**TDEventServiceInstanceEvent**
Event Type	**adaptiveEventReceived**
Code Location	invocation of the handler registered with `Event::SetReceiveHandler()` (SWS_CM_00181)
Example Code Location in APD	`ara-api/com/include/public/ara/com/internal/ dds_idl/event_data_reader_listener.h EventDataReaderListener::on_data_available(): data_Callback_();`
DltMessage	n/a
Remarks	The example code location refers to DDS binding

Table B.25: Mapping of adaptiveEventReceived

TIMEX Event	**TDEventServiceInstanceEvent**
Event Type	**adaptiveEventSent**
Code Location	invocation of the Event::Send() method (SWS_CM_00162)
Example Code Location in APD	`ara-api/com/include/public/ara/com/internal/ skeleton/event_dispatcher.h EventDispatcher::Send();`
DltMessage	n/a

Table B.26: Mapping of adaptiveEventSent

TIMEX Event	**TDEventServiceInstanceMethod**
Event Type	**adaptiveMethodCalled**
Code Location	invocation of the ServiceInterface method (SWS_CM_00196)
Example Code Location in APD	`ara-api/com/include/public/ara/com/internal/ vsomeip/proxy/vsomeip_method_impl.h ara::core::Future<> operator()()`
DltMessage	n/a
Remarks	The example code location refers to SOME/IP binding

Table B.27: Mapping of adaptiveMethodCalled

TIMEX Event	**TDEventServiceInstanceMethod**
Event Type	**adaptiveMethodCallReceived**
Code Location	invocation of the method of the service
Example Code Location in APD	`ara-api/com/include/public/ara/com/internal/ vsomeip/skeleton/vsomeip_service_impl_base.h void HandleCall(): UnmarshalAndCall()`
DltMessage	n/a
Remarks	The example code location refers to SOME/IP binding

Table B.28: Mapping of adaptiveMethodCallReceived

TIMEX Event	**TDEventServiceInstanceMethod**
Event Type	**adaptiveMethodResponseReceived**
Code Location	return of the ServiceInterface method
Example Code Location in APD	`ara-api/core/core-types/include/public/ara/core/future.h` `T get(): return GetResult().ValueOrThrow();`
DltMessage	n/a
Remarks	Besides the Example Code Location, there are other return paths to consider.

Table B.29: Mapping of adaptiveMethodResponseReceived

TIMEX Event	**TDEventServiceInstanceMethod**
Event Type	**adaptiveMethodResponseSent**
Code Location	return of the method of the service

Example Code Location in APD	`ara-api/com/include/public/ara/com/internal/`
	`vsomeip/skeleton/vsomeip_service_impl_base.h`
	`void HandleCall(): future.GetResult()`

DltMessage	n/a

Table B.30: Mapping of adaptiveMethodResponseSent

C LET Interval Constraints

C.1 Introduction

The LET interval is an important specification mechanism, as the LET paradigm allows to decouple the data flow of an application from its implementation (ref. Section 2.3.1). Beside ensuring data-flow determinism, LET also allows to design composable timing, which means that the timing behavior of data inputs and data outputs is preserved (i.e. takes place exactly at the LET events) during composition, as long as there is an implementation that satisfies the latency constraints. Since the LET paradigm can be applied already in early design stages (ref. use-case 8.2, platform independent) down to the software and network integration, it provides a powerful tool to unify the specification of timing behavior in an AUTOSAR development process.

Consequently, it is important to discuss the implications and implicit assumptions that have been made on the conceptual bounds of LET intervals. This section therefore focuses on the upper and lower bound of an LET interval and different assumptions in the related literature.

C.2 Upper Bound for the LET interval

Depending on the referenced literature, different implicit assumptions exist regarding the upper bound of an LET interval. Typically, the period is used as an upper bound, leading to $LET < period$ or $LET = period$. On the other hand, the use-case in Section 8.2 already outlined the importance of LET for the timing specification on functional level, where the latency of functional blocks likely exceeds the period.

Originated in synchronous-reactive systems [22], the period has been a consistent bound for the LET, fulfilling the synchrony hypothesis by enforcing that there is only one job running at a time. The running job finishes (constrained by the LET) before the consecutive one is started (constrained by the period). This implies a close relation between the design model and the implementation model and has been inherited in the Giotto language [8], describing the semantics of LET.

```
initialize x;
int n := 0;
while (true) {
  int k := n + offset;
  read u at k-th clock tick;
  compute x := F(x,u);
  compute y := G(x);
  k := k + let;
  write y at k-th clock tick;
  n := n + period;
}
```

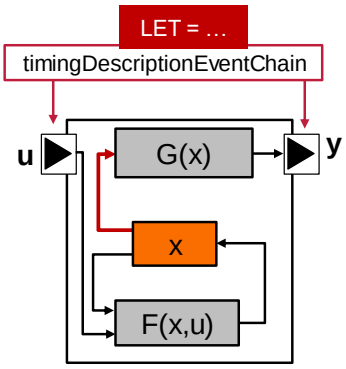

Figure C.1: Pseudo-Code of LET according to [7] with relation to AUTOSAR SWC specification

Figure C.1 shows a pseudo code of an AUTOSAR SWC that shall follow the LET paradigm, adapted from [7]. The LET specification is associated with the input and output ports of a SWC by means of a `TimingDescriptionEventChain`. For this input-output relation, the LET interval may be an arbitrary time duration. Unfortunately, a common SWC may also consist of an internal state machine, which is modeled as a Moore machine in Figure C.1. This internal state machine is hidden behind the LET, leading to an ambiguity during integration. As a consequence, the original LET paradigm is not simply applicable for $LET > period$.

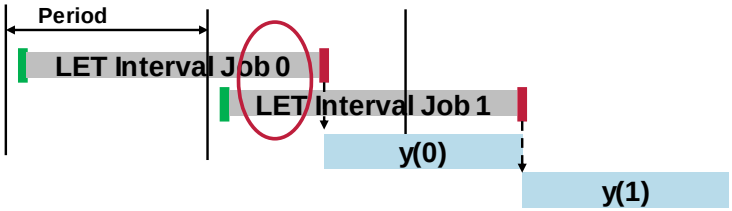

Figure C.2: Unprotected state machine for $LET > period$

Figure C.2 shows the ambiguity for the example from Figure C.1 in case $LET > period$. Multiple jobs of a SWC may be active concurrently, meaning that the next state x is used as an input for $F(x)$ and $G(x)$, before it is available. The output port still provides new samples periodically at the end of each LET interval, but without deeper insight into the SWC, the internal state and therefore the meaning of the output remains ambiguous. The output has to be assumed as generally incorrect, implementation dependent, and possibly violates the assumption of data-flow determinism.

This inconsistency can be avoided by considering one of the two following workarounds:

1. A LET specification with $LET > period$ is consistent, if the schedulable entity implementing the LET interval does not comprise an internal state that can influence the current or future outputs. This holds for many communication scenarios where data is transferred without preserving an internal state.

2. If such an internal state can not be completely avoided, the LET interval has to be partitioned in smaller sub LET intervals, such that the requirement $LET \leq period$

is met for each element containing a state. This is a concept well known from processing pipelines, where an early read on an incomplete write operation is called a data hazard.

The *Permitted Pipelining Property (PPP)* is a property of a schedulable entity in SL-LET (ref. Section 8.1), that condenses both statements:

"pipelined execution of schedulable entities in an LET interval with $LET > period$ is permitted, if no data from schedulable entities in an unfinished preceding LET interval is accessed".

This also covers cases where an LET interval abstracts a whole chain of schedulable entities (pipelining). An example is the interconnect LET, which may require schedulable entities representing basic software on the sender and the receiver ECU, as well as schedulable entities representing packets on the network. Figure C.3 provides an example to highlight the difference between permitted and prohibited data access in such a case. If the implementation would allow a data flow following one of the red dashed arrows, the data-flow determinism gained by using LET would be completely jeopardized. Therefore such cases have to be prohibited by design, selectively enabling the intended green data accesses.

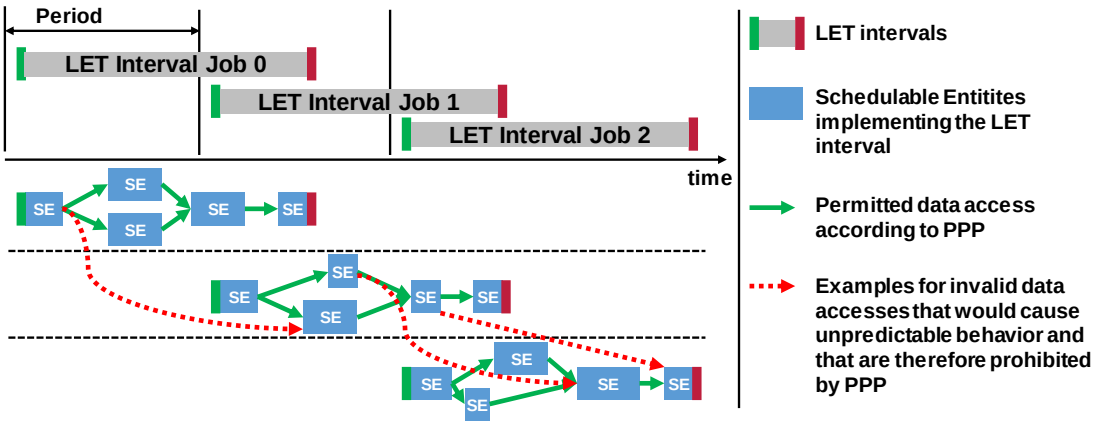

Figure C.3: Permitted and prohibited pipelining for $LET > period$

The partitioning challenge is to decompose the chain into subchains that follow either the Permitted Pipelining Property (PPP) or meet the $LET \leq period$ requirement. This also holds for the decomposition of function chains as shown in use-case 8.2 and 8.3, where an end-to-end LET interval can be partitioned in communication LET intervals (without internal state) and computation LET intervals (e.g. addressing a dedicated SWC which potentially has an internal state).

C.3 Lower Bound for the LET interval

Analogous to the upper bound of an LET interval, there may also be different motivations affecting the lower bound of an LET interval. In general, the physical execution resp. physical communication (response time) has to be able to complete during the

LET interval. Consequently this can be expressed as a formally derived WCRT. On the other hand, there are use-cases where an analytical upper bound for the response time is not feasible such as high-performance processor architectures including complex cache and execution level hierarchies. In that case, LET is even more a valuable paradigm, as the specification of LET intervals can also be done based on measured or simulated response times as well as empirical values, while the adherence to the specified behavior can be monitored with minimal overhead.

An important aspect is that LET allows to explicitly weight shorter latencies against higher robustness to system modifications. While an LET interval close to the WCRT can be used to reduce the end-to-end latency of a cause-effect chain, a later system modification may increase the WCRT. Such modifications, for example due to software updates, may originate in other functions that have a timing dependency but not a functional relationship (e.g. two SWCs from different vendors that are integrated on the same processor core). This would require an additional and potential costly design iteration to re-formulate an adapted LET interval which may have an effect on the original data flow. In contrast to that, increasing the robustness margin - namely the difference between LET interval length and the WCRT - allows the designer to introduce robustness to system modifications already in an early design stage. The remaining robustness margin can be checked for each system modification and as long as the WCRT does not violate the LET interval, the initially specified data flow is not affected.

D Composability of different implementations of (System Level-)LET

(SL-)LET serves as a specification tool in AUTOSAR. It describes an abstract time interval where data is processed (in case of LET) or processed/transferred (in case of SL-LET).[1] The interval is bounded by two time instants where communication takes place in logically zero time. The LET specification model is an *engineering model*, in the sense, that it aims to provide the maximum degree of freedom for the implementation. A "zero-time communication" can not be implemented directly, but different implementations of LET exist, which reproduce the behavior of the LET specification. To be valid, an implementation of LET has to follow two constraints. First, execution/-computation is restricted to stay within the borders of the LET interval (execution time abstraction) and second, the implementation has to provide the same data flow among jobs, as it is demanded by the LET model.

From the specification perspective, LET fulfills the requirements of a composable and platform independent timing and data flow. However, different implementations of LET have different properties with respect to composability and interoperability. This becomes problematic as soon as one LET specification is realized by different implementation styles, e.g., different suppliers. To be able to assess the composability of two implementation styles, each implementation style has to be characterized in detail (in contrast to the LET specification, the description of an implementation style is a *scientific model*).

In general, LET implementations can be characterized by borrowing terms known from synchronous digital circuit technology, namely the use of *edge-triggered flip-flops* and *level-triggered latches*. Flip-flops and latches are basic elements in synchronous digital circuits and their application allows the designer to validate the consistency of a digital circuit with simple rules. Such design rules (which are related to a scientific model) enable the scalability, portability and proven reliability of digital circuit elements in billions of components. Due to the close relation between the LET concept and synchronous digital circuit design, it is beneficial to adapt the well-known concepts when assessing LET implementations. An edge-triggered flip-flop samples the input value at an edge of a clock signal, which is, for idealistic rectangular clock signals, done exactly at a specific time instant. In contrast to that, a latch samples the input during an interval where the latch gate is open, and preserves the state when the latch gate is closed. Both have in common that there exist non-zero setup- and hold-times, during which a state change has to be prohibited. D.1 provides an example how this analogy can be applied to an implementation model of LET. A label (which is stored in a memory) can be seen as a digital wire, where consumers may be attached. Each read or write access to the wire is gated, either by a flip-flop or by a latch. The main difference to synchronous digital circuit technology is, that each gate has its individual clock signal which reflects the periodic LET events for that gate. Beside uniform implementation styles, that only apply latches or flip-flops on both sides, hybrid approaches may be possible and one have to take care how different implementations may interact.

[1] In the following, only LET is referred, but the idea applies to both, LET and SL-LET.

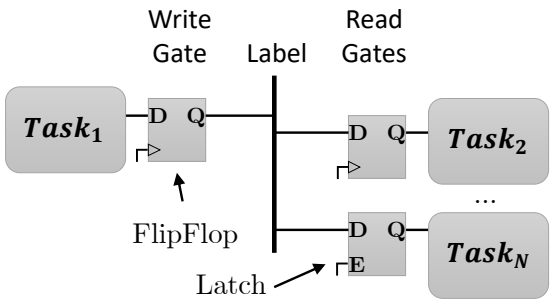

Figure D.1: Different LET implementation styles: Access to a shared label (digital wire) with flip-flops or latches

A scientific model of a LET implementation style covers at least the following points:

1. When is the data accessed (when is the logical gate open)?
2. How is the data access related to the LET interval? May it actually take place outside the borders of the LET interval?
3. What is the time/clock reference? Is there a synchronization accuracy that has to be covered?
4. How much clearance (setup- and hold-time) is required?
5. Are there any additional synchronization mechanisms involved (e.g., spinlocks)?
6. Which type of communication (explicit or implicit) is assumed by the application?

Figure D.2 provides an example what might happen if a LET specification is realized by two LET implementation styles. In this example, one core of a multicore ECU uses a dedicated LET implementation style, e.g., provided by a different supplier. The LET intervals in the specification are placed back-to-back, meaning that $LET_{2,3,4}$ will read from LET_1. Therefore the LET events fall to the same time instant and the write-before-read consistency needs to be guaranteed. In an implementation, those time instants are typically represented by a timer interrupt. In the example in Figure D.2, this interrupt triggers all processor cores, but the first implementation style then starts to process first all write and then all read operations. Although the write operation W_1 now takes place outside the borders of the LET interval LET_1, the implementation ensures consistency by synchronizing the processor cores with a spinlock. If the second implementation of LET does not participate in this synchronization mechanism, the consistency might be violated. Without touching the implementation, an option would be to adapt the LET specification. This can be either by adding an offset to the LET_4 interval or by enforcing that the write operation W_1 actually takes place within the borders of the LET interval LET_1.

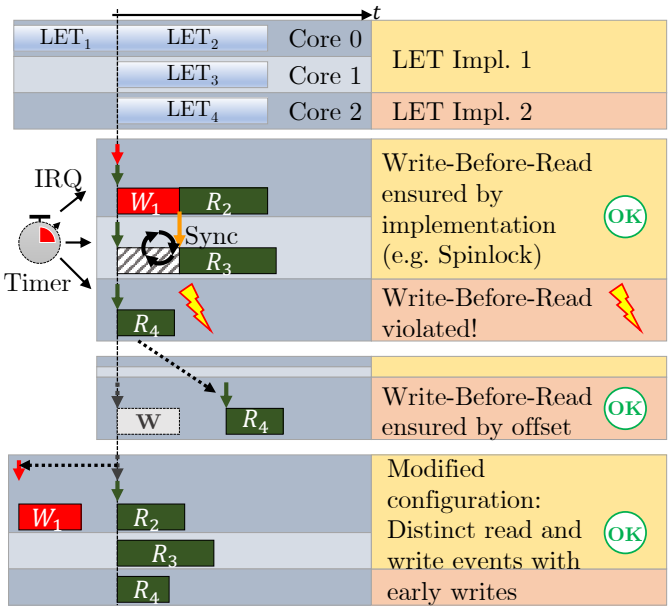

Figure D.2: Ensuring consistency between different implementations of LET

E History of Constraints and Specification Items

E.1 Constraint History of this Document related to AUTOSAR R4.1.3

E.1.1 Changed Constraints in R4.1.3

No constraints were changed in this release.

E.1.2 Added Constraints in R4.1.3

No constraints were added in this release.

E.1.3 Deleted Constraints in R4.1.3

No constraints were deleted in this release.

E.2 Specification Items History of this Document related to AUTOSAR R4.1.3

E.2.1 Changed Specification Items in R4.1.3

No specification items were changed in this release.

E.2.2 Added Specification Items in R4.1.3

No specification items were added in this release.

E.2.3 Deleted Specification Items in R4.1.3

No specification items were deleted in this release.

F List of figures, list of tables, and index

List of Figures

List of Tables

Index